

2nd Edition

FUNDAMENTALS OF DATA STRUCTURES IN C++

HOROWITZ ☆ SAHNI ☆ MEHTA

Subhojit

***Fundamentals of
Data Structures
in C++***

Second Edition

Ellis Horowitz

University of Southern California

Sartaj Sahni

University of Florida

Dinesh P. Mehta

Colorado School of Mines

Contents

CHAPTER 1	BASIC CONCEPTS	1
1.1	Overview: System Life Cycle	1
1.2	Object-Oriented Design	4
1.2.1	Algorithmic Decomposition versus OO Decomposition	5
1.2.2	Fundamental Definitions and Concepts of OO Programming	5
1.2.3	Evolution of Programming Languages and History of C++	6
1.3	Data Abstraction and Encapsulation	7
1.4	Basics of C++	12
1.4.1	Program Organization in C++	13
1.4.2	Scope in C++	13
1.4.3	C++ Statements and Operators	15
1.4.4	Data Declarations in C++	15
1.4.5	Comments in C++	16
1.4.6	Input/Output in C++	17
1.4.7	Functions in C++	18
1.4.8	Parameter Passing in C++	19

1.4.9	Function Name Overloading in C++	21
1.4.10	Inline Functions	21
1.4.11	Dynamic Memory Allocation in C++	21
1.4.12	Exceptions	22
1.5	Algorithm Specification	25
1.5.1	Introduction	25
1.5.2	Recursive Algorithms	29
1.6	The Standard Template Library	34
1.7	Performance Analysis And Measurement	37
1.7.1	Performance Analysis	38
1.7.2	Performance Measurement	61
1.7.3	Generating Test Data	69
1.8	References And Selected Readings	73
 CHAPTER 2 ARRAYS 74		
2.1	Abstract Data Types and the C++ Class	74
2.1.1	An Introduction to the C++ Class	74
2.1.2	Data Abstraction and Encapsulation in C++	75
2.1.3	Declaring Class Objects and Invoking Member Functions	76
2.1.4	Special Class Operations	77
2.1.5	Miscellaneous Topics	81
2.1.6	ADTs and C++ Classes	82
2.2	The Array as an Abstract Data Type	84
2.3	The Polynomial Abstract Data Type	86
2.3.1	Polynomial Representation	86
2.3.2	Polynomial Addition	90
2.4	Sparse Matrices	95
2.4.1	Introduction	95
2.4.2	Sparse Matrix Representation	97
2.4.3	Transposing a Matrix	99
2.4.4	Matrix Multiplication	102
2.5	Representation of Arrays	108
2.6	The String Abstract Data Type	113
2.6.1	String Pattern Matching: A Simple Algorithm	114
2.6.2	String Pattern Matching: The KMP Algorithm	115
2.7	References and Selected Readings	120
2.8	Additional Exercises	120
 CHAPTER 3 STACKS AND QUEUES 128		
3.1	Templates in C++	128

3.1.1	Template Functions	128
3.1.2	Using Templates to Represent Container Classes	130
3.2	The Stack Abstract Data Type	134
3.3	The Queue Abstract Data Type	139
3.4	Subtyping and Inheritance in C++	148
3.5	A Mazing Problem	152
3.6	Evaluation of Expressions	157
3.6.1	Expressions	157
3.6.2	Postfix Notation	161
3.6.3	Infix to Postfix	162
3.7	Additional Exercises	167
CHAPTER 4	LINKED LISTS	170
4.1	Singly Linked Lists and Chains	170
4.2	Representing Chains in C++	173
4.2.1	Defining a Node in C++	173
4.2.2	Designing a Chain Class in C++	176
4.2.3	Pointer Manipulation in C++	179
4.2.4	Chain Manipulation Operations	181
4.3	The Template Class Chain	185
4.3.1	Implementing Chains with Templates	185
4.3.2	Chain Iterators	185
4.3.3	Chain Operations	190
4.3.4	Reusing a Class	193
4.4	Circular Lists	194
4.5	Available Space Lists	197
4.6	Linked Stacks and Queues	199
4.7	Polynomials	202
4.7.1	Polynomial Representation	202
4.7.2	Adding Polynomials	204
4.7.3	Circular List Representation of Polynomials	207
4.8	Equivalence Classes	210
4.9	Sparse Matrices	216
4.9.1	Sparse Matrix Representation	216
4.9.2	Sparse Matrix Input	219
4.9.3	Deleting a Sparse Matrix	220
4.10	Doubly Linked Lists	224
4.11	Generalized Lists	228
4.11.1	Representation of Generalized Lists	228
4.11.2	Recursive Algorithms for Lists	232
4.11.3	Reference Counts, Shared and Recursive Lists	236

CHAPTER 5	TREES	243
5.1	Introduction	243
5.1.1	Terminology	243
5.1.2	Representation of Trees	246
5.2	Binary Trees	251
5.2.1	The Abstract Data Type	251
5.2.2	Properties of Binary Trees	251
5.2.3	Binary Tree Representations	253
5.3	Binary Tree Traversal and Tree Iterators	259
5.3.1	Introduction	259
5.3.2	Inorder Traversal	260
5.3.3	Preorder Traversal	260
5.3.4	Postorder Traversal	261
5.3.5	Iterative Inorder Traversal	262
5.3.6	Level-Order Traversal	265
5.3.7	Traversal Without a Stack	266
5.4	Additional Binary Tree Operations	269
5.4.1	Copying Binary Trees	269
5.4.2	Testing Equality	269
5.4.3	The Satisfiability Problem	269
5.5	Threaded Binary Trees	274
5.5.1	Threads	274
5.5.2	Inorder Traversal of a Threaded Binary Tree	276
5.5.3	Inserting a Node into a Threaded Binary Tree	277
5.6	Heaps	279
5.6.1	Priority Queues	279
5.6.2	Definition of a Max Heap	281
5.6.3	Insertion into a Max Heap	283
5.6.4	Deletion from a Max Heap	283
5.7	Binary Search Trees	287
5.7.1	Definition	287
5.7.2	Searching a Binary Search Tree	289
5.7.3	Insertion into a Binary Search Tree	291
5.7.4	Deletion from a Binary Search Tree	291
5.7.5	Joining and Splitting Binary Search Trees	293
5.7.6	Height of a Binary Search Tree	294
5.8	Selection Trees	297
5.8.1	Introduction	297
5.8.2	Winner Trees	297
5.8.3	Loser Trees	300
5.9	Forests	301
5.9.1	Transforming a Forest into a Binary Tree	307

5.9.2	Forest Traversals	303
5.10	Representation of Disjoint Sets	304
5.10.1	Introduction	304
5.10.2	Union and Find Operations	305
5.10.3	Application to Equivalence Classes	314
5.11	Counting Binary Trees	317
5.11.1	Distinct Binary Trees	317
5.11.2	Stack Permutations	317
5.11.3	Matrix Multiplication	320
5.11.4	Number of Distinct Binary Trees	321
5.12	References and Selected Readings	323
CHAPTER 6	GRAPHS	324
6.1	The Graph Abstract Data Type	324
6.1.1	Introduction	324
6.1.2	Definitions	325
6.1.3	Graph Representations	330
6.2	Elementary Graph Operations	340
6.2.1	Depth First Search	341
6.2.2	Breadth First Search	343
6.2.3	Connected Components	344
6.2.4	Spanning Trees	344
6.2.5	Biconnected Components	347
6.3	Minimum Cost Spanning Trees	352
6.3.1	Kruskal's Algorithm	353
6.3.2	Prim's Algorithm	357
6.3.3	Sollin's Algorithm	359
6.4	Shortest Paths and Transitive Closure	360
6.4.1	Single Source/All Destination: Nonnegative Edge Costs	361
6.4.2	Single Source/All Destination: General Weights	364
6.4.3	All-Pairs Shortest Paths	368
6.4.4	Transitive Closure	370
6.5	Activity Networks	375
6.5.1	Activity on Vertex (AOV) Networks	375
6.5.2	Activity on Edge (AOE) Networks	381
6.6	References and Selected Readings	390
6.7	Additional Exercises	390
CHAPTER 7	SORTING	394
7.1	Motivation	394

xviii Contents

7.2	Insertion Sort	399	
7.3	Quick Sort	402	
7.4	How Fast Can We Sort?	405	
7.5	Merge Sort	407	
	7.5.1 Merging	407	
	7.5.2 Iterative Merge Sort	409	
	7.5.3 Recursive Merge Sort	410	
7.6	Heap Sort	414	
7.7	Sorting on Several Keys	417	
7.8	List and Table Sorts	423	
7.9	Summary of Internal Sorting	432	
7.10	External Sorting	438	
	7.10.1 Introduction	438	
	7.10.2 k-way Merging	442	
	7.10.3 Buffer Handling for Parallel Operation	444	
	7.10.4 Run Generation	450	
	7.10.5 Optimal Merging of Runs	453	
7.11	References and Selected Readings	457	
 CHAPTER 8 HASHING 458			
8.1	Introduction	458	
8.2	Static Hashing	459	
	8.2.1 Hash Tables	459	
	8.2.2 Hash Functions	461	
	8.2.3 Secure Hash Functions	464	
	8.2.4 Overflow Handling	467	
	8.2.5 Theoretical Evaluation of Overflow Techniques	473	
8.3	Dynamic Hashing	477	
	8.3.1 Motivation for Dynamic Hashing	477	
	8.3.2 Dynamic Hashing using Directories	478	
	8.3.3 Directoryless Dynamic Hashing	480	
8.4	Bloom Filters	483	
	8.4.1 An Application -- Differential Files	483	
	8.4.2 Bloom Filter Design	486	
8.5	References and selected Readings	488	
 CHAPTER 9 PRIORITY QUEUES 489			
9.1	Single- and Double-Ended Priority Queues	489	
9.2	Leftist Trees	492	
	9.2.1 Height-Biased Leftist Trees	492	
	9.2.2 Weight-Biased Leftist Trees	499	

9.3	Binomial Heaps	502
9.3.1	Cost Amortization	502
9.3.2	Definition of Binomial Heaps	503
9.3.3	Insertion into a Binomial Heap	504
9.3.4	Melding Two Binomial Heaps	505
9.3.5	Deletion of Min Element	506
9.3.6	Analysis	509
9.4	Fibonacci Heaps	512
9.4.1	Definition	512
9.4.2	Deletion from an F-heap	513
9.4.3	Decrease Key	513
9.4.4	Cascading Cut	514
9.4.5	Analysis	515
9.4.6	Application to The Shortest Paths Problem	518
9.5	Pairing Heaps	520
9.5.1	Definition	520
9.5.2	Meld and Insert	522
9.5.3	Decrease Key	522
9.5.4	Delete Min	524
9.5.5	Arbitrary Delete	526
9.5.6	Implementation Considerations	527
9.5.7	Complexity	528
9.6	Symmetric Min-Max Heaps	529
9.6.1	Definition and Properties	529
9.6.2	SMMH Representation	530
9.6.3	Inserting into an SMMH	531
9.6.4	Deleting from an SMMH	537
9.7	Interval Heaps	541
9.7.1	Definition and Properties	541
9.7.2	Inserting into an Interval Heap	543
9.7.3	Deleting the Min Element	545
9.7.4	Initializing an Interval Heap	547
9.7.5	Complexity of Interval Heap Operations	547
9.7.6	The Complementary Range Search Problem	548
9.8	References and Selected Readings	551
CHAPTER 10 EFFICIENT BINARY SEARCH TREES		553
10.1	Optimal Binary Search Trees	553
10.2	AVL Trees	564
10.3	Red-Black Trees	579
10.3.1	Definition	579
10.3.2	Representation of a Red-Black Tree	581

10.3.3	Searching a Red-Black Tree	581
10.3.4	Inserting into a Red-Black Tree	581
10.3.5	Deletion from a Red-Black Tree	586
10.3.6	Joining Red-Black Trees	586
10.3.7	Splitting a Red-Black Tree	588
10.4	Splay Trees	592
10.4.1	Bottom-Up Splay Trees	592
10.4.2	Top-Down Splay Trees	598
10.5	References and Selected Readings	605

CHAPTER 11 MULTIWAY SEARCH TREES 606

11.1	m -way Search Trees	606
11.1.1	Definition and Properties	606
11.1.2	Searching an m -way Search Tree	608
11.2	B-Trees	609
11.2.1	Definition and Properties	609
11.2.2	Number of Elements in a B-Tree	611
11.2.3	Insertion into a B-tree	613
11.2.4	Deletion from a B-tree	615
11.3	B^+ -Trees	626
11.3.1	Definition	626
11.3.2	Searching a B^+ -Tree	627
11.3.3	Insertion into a B^+ -Tree	628
11.3.4	Deletion from a B^+ -Tree	629
11.4	References and Selected Readings	635

CHAPTER 12 DIGITAL SEARCH STRUCTURES 637

12.1	Digital Search Trees	637
12.1.1	Definition	637
12.1.2	Search, Insert and Delete	638
12.2	Binary Tries and Patricia	639
12.2.1	Binary Tries	639
12.2.2	Compressed Binary Tries	640
12.2.3	Patricia	640
12.3	Multiway Tries	647
12.3.1	Definition	647
12.3.2	Searching a Trie	649
12.3.3	Sampling Strategies	649
12.3.4	Insertion into a Trie	653
12.3.5	Deletion from a Trie	653
12.3.6	Keys with Different Length	654

12.3.7	Height of a Trie	655	
12.3.8	Space Required and Alternative Node Structures	655	
12.3.9	Prefix Search and Applications	659	
12.3.10	Compressed Tries	660	
12.3.11	Compressed Tries with Skip Fields	664	
12.3.12	Compressed Tries with Labeled Edges	664	
12.3.13	Space Required by a Compressed Trie	669	
12.4	Suffix Trees	670	
12.4.1	Have You Seen this String?	670	
12.4.2	The Suffix Tree Data Structure	671	
12.4.3	Let's Find That Substring (Searching a Suffix Tree)	674	
12.4.4	Other Nifty Things You Can Do with a Suffix Tree	676	
12.5	Tries and Internet Packet Forwarding	678	
12.5.1	IP Routing	678	
12.5.2	1-Bit Tries	679	
12.5.3	Fixed-Stride Tries	680	
12.5.4	Variable-Stride Tries	683	
12.6	References and Selected Readings	686	

INDEX	688
-------	-----

CHAPTER 1

Basic Concepts

1.1 OVERVIEW: SYSTEM LIFE CYCLE

We assume that our readers have a strong background in programming, typically attained through the completion of an elementary programming course. Such an initial course usually emphasizes mastering the syntax of a programming language (its grammar rules) and applying this language to the solution of several relatively small problems. In this text we want to move beyond these rudiments by providing the tools and techniques necessary to design and implement large-scale computer systems. We believe that a solid foundation in data abstraction and encapsulation, algorithm specification, and performance analysis and measurement provides the necessary methodology. In this chapter, we will discuss each of these areas in detail. We will also briefly discuss recursive programming because many of you probably have only a fleeting acquaintance with this important technique. However, before we begin, we want to place these tools in a context that views programming as more than writing code. Good programmers regard large-scale computer programs as systems that contain many complex interacting parts. As systems, these programs undergo a development

2 Basic Concepts

process called the system life cycle. This cycle consists of requirements, analysis, design, coding, and verification phases. Although we will consider them separately, these phases are highly interrelated and follow only a very crude sequential time frame. The References and Selected Readings section lists several sources on the system life cycle and its various phases that will provide you with additional information.

(1) **Requirements.** All large programming projects begin with a set of specifications that define the purpose of the project. These requirements describe the information that we, the programmers, are given (input) and the results that we must produce (output). Frequently the initial specifications are defined vaguely, and we must develop rigorous input and output descriptions that include all cases.

(2) **Analysis.** After we have delineated carefully the system's requirements, the analysis phase begins in earnest. In this phase, we begin to break the problem down into manageable pieces. There are two approaches to analysis: bottom-up and top-down. The bottom-up approach is an older, unstructured strategy that places an early emphasis on the coding fine points. Since the programmer does not have a master plan for the project, the resulting program frequently has many loosely connected, error-ridden segments. Bottom-up analysis is akin to constructing a building by first designing specific aspects of the building such as walls, a roof, plumbing, and heating, and then trying to put these together to construct the building. The specific purpose to which the building will be put is not considered in this approach. Although few of us would want to live in a home constructed using this technique, many programmers, particularly beginning ones, believe that they can create good, error-free programs without prior planning.

In contrast, the top-down approach begins by developing a high-level plan for dividing the program into manageable segments. This plan is subsequently refined to take into account low-level details. This technique generates diagrams that are used to design the system. Frequently, several alternate solutions to the programming problem are developed and compared during this phase. The top-down approach has been the preferred approach for developing complex software systems.

(3) **Design.** This phase continues the work done in the analysis phase. The designer approaches the system from the perspectives of the data objects that the program needs and the operations performed on them. The first perspective leads to the creation of abstract data types, whereas the second requires the specification of algorithms and a consideration of algorithm design strategies. For example, suppose that we are designing a scheduling system for a university.

Typical data objects might include students, courses, and professors. Typical operations might include inserting, removing, and searching within each object or between them. That is, we might want to add a course to the list of university courses or search for the courses taught by a specific professor.

Since the abstract data types and the algorithm specifications are language-independent, we postpone implementation decisions. Although we must specify the information required for each data object, we ignore coding details. For example, we might decide that the student data object should include name, social security number, major, and phone number. However, we would not yet pick a specific implementation for the list of students. As we will see in later chapters, there are several possibilities, including arrays, linked lists, or trees. By deferring implementation issues as long as possible, we not only create a system that could be written in several programming languages, but we also have time to pick the most efficient implementations within our chosen language.

(4) **Refinement and coding.** In this phase, we choose representations for our data objects and write algorithms for each operation on them. The order in which we do this is crucial because a data object's representation can determine the efficiency of the algorithms related to it. Typically this means that we should write the algorithms that are independent of the data objects first.

Frequently at this point we realize that we could have created a much better system. Perhaps we have spoken with a friend who has worked on a similar project, or we realize that one of our alternate designs is superior. If our original design is good, it can absorb changes easily. In fact, this is a reason for avoiding an early commitment to coding details. If we must scrap our work entirely, we can take comfort in the fact that we will be able to write the new system more quickly and with fewer errors.

(5) **Verification.** This phase consists of developing correctness proofs for the program, testing the program with a variety of input data, and removing errors. Each of these areas has been researched extensively, and a complete discussion is beyond the scope of this text. However, we summarize briefly the important aspects of each area.

Correctness proofs: Programs can be proven correct using the same techniques that abound in mathematics. Unfortunately, these proofs are very time-consuming and difficult to develop for large projects. Frequently, scheduling constraints prevent the development of a complete set of proofs for a large system. However, selecting algorithms that have been proven correct can reduce the number of errors. In this text, we will provide you with an arsenal of algorithms, some of which have been proven correct using formal techniques, that

4 Basic Concepts

you may apply to many programming problems.

Testing: Since our algorithms need not be written in a specific programming language, we can construct our correctness proofs before and during the coding phase. Testing, however, requires the working code and sets of test data. This data should be developed carefully so that it includes all possible scenarios. Frequently, beginning programmers assume that if their program ran without producing a syntax error, it must be correct. Little thought is given to the input data, and usually only one set of data is used. Good test data should verify that every piece of code runs correctly. For example, if our program contains a `switch` statement, our test data should be chosen so that we can check each case within the `switch` statement.

Initial system tests focus on verifying that a program runs correctly. Although this is a crucial concern, a program's running time is also important. An error-free program that runs slowly is of little value. Theoretical estimates of running time exist for many algorithms. We will derive these estimates as we introduce new algorithms. In addition, we may want to gather performance estimates for portions of our code. Constructing these timing tests is also a topic that we pursue later in this chapter.

Error removal: If done properly, the correctness proofs and system tests will indicate erroneous code. The ease with which we can remove these errors depends on the design and coding decisions made earlier. A large undocumented program written in "spaghetti" code is a programmer's nightmare. When debugging such programs, each corrected error possibly generates several new errors. On the other hand, debugging a well-documented program that is divided into autonomous units that interact through parameters is far easier, especially if each unit is tested separately and then integrated into the system.

1.2 OBJECT-ORIENTED DESIGN

Object-oriented design represents a fundamental change from the structured programming design method. The two approaches are similar in that both believe that the way to develop a complex system is by using the philosophy of divide-and-conquer: that is, break up a complex software design project into a number of simpler subprojects, and then tackle these subprojects individually. The two approaches disagree on how a project should be decomposed.

1.2.1 Algorithmic Decomposition Versus Object-Oriented Decomposition

Traditional programming techniques have used algorithmic decomposition. Algorithmic or functional decomposition views software as a *process*. It decomposes the software into modules that represent steps of the process. These modules are implemented by language constructs such as procedures in Pascal, subprograms in Fortran, and functions in C. The data structures required to implement the program are a secondary concern, which is addressed after the project has been decomposed into functional modules.

Object-oriented decomposition views software as a set of well-defined *objects* that model entities in the application domain. These objects interact with each other to form a software system. Functional decomposition is addressed after the system has been decomposed into objects.

The principal advantage of object-oriented decomposition is that it encourages the reuse of software. This results in flexible software systems that can evolve as system requirements change. It allows a programmer to use object-oriented programming languages effectively. Object-oriented decomposition is also more intuitive than algorithm-oriented decomposition because objects naturally model entities in the application domain.

1.2.2 Fundamental Definitions and Concepts of Object-Oriented Programming

Before proceeding, we provide definitions of basic terminology used in the object-oriented paradigm. Our definitions are adapted from *Object-Oriented Design with Applications*, by Grady Booch.

Definition: An *object* is an entity that performs computations and has a local state. It may therefore be viewed as a combination of data and procedural elements.

Definition: *Object-oriented programming* is a method of implementation in which

- (1) Objects are the fundamental building blocks.
- (2) Each object is an instance of some type (or class).
- (3) Classes are related to each other by inheritance relationships. (Programming methods that do not use inheritance are not considered to be object-oriented.)

6 Basic Concepts

Definition: A language is said to be an *object-oriented language* if

- (1) It supports objects.
- (2) It requires objects to belong to a class.
- (3) It supports inheritance.

A language that supports the first two features but does not support inheritance, is an *object-based language*. C++ is an *object-oriented language* while JavaScript is an *object-based language*.

1.2.3 Evolution of Programming Languages and History of C++

Higher order programming languages may be classified into four generations.

- (1) **First Generation Languages:** An example of this is FORTRAN. The salient feature of these languages is their ability to evaluate mathematical expressions.
- (2) **Second Generation Languages:** Examples of these include Pascal and C. The emphasis of these languages is on effectively expressing algorithms.
- (3) **Third Generation Languages:** An example of this is Modula, which introduced the concept of abstract data types.
- (4) **Fourth Generation Languages (Object-Oriented languages):** Examples include Smalltalk, Objective C, and C++. These languages emphasize the expression of the relationship between abstract data types through the use of inheritance.

C++ was designed by Bjarne Stroustrup of AT&T Bell Laboratories in the early 1980s. It was designed to incorporate the object-oriented paradigm into the C programming language. C is widely-used in industry because it is:

- (1) **Efficient:** it has a number of low-level features, which utilize hardware better than other languages.
- (2) **Flexible:** it can be used to solve problems in most application areas.
- (3) **Available:** compilers for C are readily available for most computers.

Other influences on the design of C++ were Simula67 and Algol68.

There are a number of features of C++ that improve on C, but do not implement data abstraction or inheritance. These are discussed later in this chapter. The C++ class, which implements data abstraction, is discussed in

Chapter 2, C++ functions and classes can be made more powerful by augmenting them with templates. Templates are discussed in Chapter 3. Simple inheritance in C++ is also discussed in Chapter 3. Advanced concepts on inheritance and polymorphism are discussed in Chapter 4.

1.3 DATA ABSTRACTION AND ENCAPSULATION

The notions of abstraction and encapsulation are commonly used in human-machine interaction and are widely prevalent in the technology-oriented world we live in. Consider, for example, the digital video player (DVD player), which is an integral part of many households today. We make two observations about how DVD players are packaged and used.

(1) All our interactions with a DVD player are made through buttons on the control panel (or remote control) such as PLAY, STOP and PAUSE. The DVD player is packaged so that we cannot directly interact with the circuitry inside. So, the *internal representation* of the DVD player is *hidden* from the user. This is the principle of *encapsulation*.

(2) The instruction manual that accompanies the DVD player tells us *what* the DVD player is supposed to do if we press a particular button. It does not tell us *how* this function is implemented inside the DVD player; it does not, for example, explain the sequence of electronic and mechanical events that result when the PLAY button is pressed. So, the instruction manual makes a clear distinction between *what* the DVD player does and *how* it does it. This distinction is the principle of *abstraction*.

These same principles may be applied to the packaging or organizing of data in a computer program, resulting in the concepts of data encapsulation and data abstraction, which are defined below.

Definition: *Data Encapsulation or Information Hiding* is the concealing of the implementation details of a data object from the outside world.

Definition: *Data Abstraction* is the separation between the *specification* of a data object and its *implementation*.

We will see later that these concepts are of great importance in software development because they result in better quality programs and more efficient programming techniques. These, in turn, reduce the number of person-hours

8 Basic Concepts

required to develop a software system, and hence reduce the total cost of the final software product.

First, however, we shall explore the fundamental data types of C++. These include `char`, `int`, `float`, and `double`. Some of these data types may be modified by keywords `short`, `long`, `signed`, and `unsigned`. The modifiers `short` and `long` specify the amount of storage allocated to the fundamental type they modify. The modifiers `signed` and `unsigned` specify whether the most significant bit in the binary representation of an integer represents a sign bit or not. C++ also supports types derived from the fundamental data types listed above. These include pointer and reference types. Ultimately, the real world abstractions we wish to deal with must be represented in terms of these data types. In addition to these types, C++ helps us by providing three mechanisms for grouping data together. These are the array, the struct, and the class. Arrays are collections of elements of the same basic data type; structs and classes are collections of elements whose data types need not be the same.

All programming languages provide at least a minimal set of predefined data types, plus the ability to construct new, or user-defined types.

Definition: A data type is a collection of objects and a set of operations that act on those objects. □

Whether your program is dealing with predefined data types or user-defined data types, these two aspects must be considered: objects and operations. For example, the data type `int` consists of the objects $\{0, +1, -1, +2, -2, \dots, \text{MAXINT}, \text{MININT}\}$, where `MAXINT` and `MININT` are the largest and smallest integers that can be represented by an `int` on your machine. The operations on integers are many and would certainly include the arithmetic operators `+`, `-`, `*`, and `/`. There is also testing for equality/inequality and the operation that assigns an integer to a variable.

Definition: An abstract data type (ADT) is a data type that is organized in such a way that the specification of the objects and the specification of the operations on the objects is separated from the representation of the objects and the implementation of the operations. □

Throughout this text, we will emphasize the distinction between specification and implementation. To help us do this, we will typically begin with an ADT definition of the object that we intend to study. This will permit the reader to grasp the essential elements of the object, without having the discussion complicated by the representation of the objects or by the actual implementation of the operations. Once the ADT definition is fully explained, we will move on to discussions of representation and implementation. These are quite

important in the study of data structures. To help us accomplish this goal, we introduce a notation for expressing an ADT. This notation is independent of the syntax of the C++ language. Later we shall see how the syntax of the C++ class can be used to express an ADT.

Example 1.1 [Abstract data type *NaturalNumber*]: As this is the first example of an ADT, we will spend some time explaining the notation. ADT 1.1 contains the ADT definition of *NaturalNumber*. The definition begins with the name of the abstract data type. There are two main sections in the definition: the objects and the functions. The objects are defined in terms of the integers, but we make no explicit reference to their representation. The function definitions are a bit more complicated. First, the definitions use the symbols x and y to denote two elements of the set *NaturalNumber*, whereas TRUE and FALSE are elements of the set of *Boolean* values. In addition, the definition makes use of functions that are defined on the set of integers, namely, plus, minus, equal, and less than. This is an indication that in order to define one data type, we may need to use operations from another data type. For each function, we place the result type to the left of the function name and a definition of the function to the right. The symbols " ::= " should be read as "is defined as."

The first function, *Zero*, has no arguments and returns the natural number zero. The function *Successor(x)* returns the next natural number in sequence. Notice that if there is no next number in sequence, that is, if the value of x is already MAXINT, then we define the action of *Successor* to return MAXINT. Some programmers might prefer that in such a case *Successor* return an error flag. This is also perfectly permissible. Other functions are *Add* and *Subtract*. They might also return an error condition, although here we decided to return an element of the set *NaturalNumber*. □

Next, we see how the principles of data abstraction and data encapsulation help us to efficiently develop well-designed programs.

(1) **Simplification of software development:** The chief advantage of data abstraction is that it facilitates the decomposition of the complex task of developing a software system into a number of simpler subtasks. Consider the following scenario: A problem has been assigned to a single programmer or to a team of programmers. Suppose that a top-down review of the problem results in a decision that three data types A, B, and C will be used along with some additional code (which we will call *glue*) to facilitate interactions among the three data types. Assume, also, that the specifications of each data type are provided at the outset; i.e., the operations that each data type must support are completely specified.

10 Basic Concepts

ADT *NaturalNumber* is

objects: An ordered subrange of the integers starting at zero and ending at the maximum integer (MAXINT) on the computer.

Functions:

for all $x, y \in \text{NaturalNumber}$; TRUE, FALSE \in Boolean
and where +, -, <, ==, and = are the usual integer operations

<i>Zero</i> (): <i>NaturalNumber</i>	==	0
<i>IsZero</i> (x): Boolean	==	if ($x == 0$) <i>IsZero</i> = true else <i>IsZero</i> = false
<i>Add</i> (x, y): <i>NaturalNumber</i>	==	if ($x + y <= \text{MAXINT}$) <i>Add</i> = $x + y$ else <i>Add</i> = MAXINT
<i>Equal</i> (x, y): Boolean	==	if ($x == y$) <i>Equal</i> = TRUE else <i>Equal</i> = FALSE
<i>Successor</i> (x): <i>NaturalNumber</i>	==	if ($x == \text{MAXINT}$) <i>Successor</i> = x else <i>Successor</i> = $x + 1$
<i>Subtract</i> (x, y): <i>NaturalNumber</i>	==	if ($x < y$) <i>Subtract</i> = 0 else <i>Subtract</i> = $x - y$

end *NaturalNumber*

ADT 1.1: Abstract data type *NaturalNumber*

(a) **Scenario 1 - A team of four programmers:** the subtasks may be allotted to members of a team of four programmers as follows. A programmer is assigned to each of the three data types. Each programmer's job is to implement his/her data type according to the specifications agreed upon earlier. The fourth programmer implements the glue, assuming that the data types are, in fact, implemented according to the specifications. No programmer needs to know how the other programmers implement their portion of the code. Consequently, each programmer can focus exclusively on his/her portion of the code without worrying about what the other programmers are doing and how that might affect what he/she is doing.

(b) **Scenario 2 - A single programmer:** here data abstraction helps reduce the number of things the programmer has to keep in mind at any time. In the example, the programmer would implement each data type one by one according to the specifications. When the programmer implements data type A, he/she can do so without worrying about data types B and C and the glue. After the individual data types are implemented, the programmer can turn his/her attention to the glue. At this point, the programmer is no longer concerned with the

implementations of the three data types, but rather with how to use operations on them to achieve the desired goal.

(2) **Testing and Debugging:** Breaking down a complex task into a number of easier subtasks also simplifies testing and debugging. In the above example, each data type can be tested and debugged separately. After each data type is tested and debugged, the entire program can be tested. If a bug is detected at this stage, it is unlikely that there is a bug in the data structures *A*, *B*, and *C* (represented by the shaded boxes of Figure 1.1(a)) because they have already been tested. So, the bug will most likely be in the glue represented by the unshaded portions of Figure 1.1(a). Compare this with the amount of code to be searched if we had *not* used data abstraction (represented by the unshaded portions Figure 1.1(b)). It is easy to see that employing data abstraction in software development leads to more efficient testing and debugging.

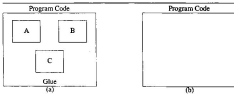


Figure 1.1: Unshaded areas represent code that has to be searched for bugs: (a) data abstraction is used (b) data abstraction is not used.

(3) **Reusability:** Data abstraction and encapsulation typically give rise to data structures that are implemented as distinct entities of a software system. This makes it easier to extract the code for a data structure and its operations from a software system and use it in another software system, rather than if it were inextricably integrated with the original software system. We will later discuss other techniques in C++ that enhance the reusability of software.

(4) **Modifications to the representation of a data type:** a significant

12 Basic Concepts

consequence of information-hiding is that the implementation of a data type is not visible to the rest of the program. Specifically, the rest of the program does not have direct access to the internal representation of the data type. It manipulates the data type solely through a suite of operations that the data type has made available to it. Thus, a change in the internal implementation of a data type will not affect the rest of the program as long as the data type continues to provide the same operations, and as long as these operations continue to do the same things. All that is required is a change in the implementation of the operations that directly access the internal implementation of the data type.

This is easier to appreciate if we examine the consequences of *not* employing data encapsulation. Consider a program that directly manipulates the internal representation of its data types. Suppose a change is made to the implementation of a data type. You, the programmer, are now confronted with the tasks of first examining the program code to locate instances where the program accesses the internal representation of the data type and then making the appropriate changes. This could be an extremely laborious task if the program accesses the internal representation of the data type many times.

EXERCISES

For each of these exercises, provide a definition of the abstract data type using the form illustrated in ADT 1.1.

1. Add the following operations to the *NaturalNumber* ADT: *Predecessor*, *IsGreater*, *Multiply*, *Divide*.
2. Create an ADT, *Set*. Use the standard mathematical definition and include the following operations: *Create*, *Insert*, *Remove*, *IsIn*, *Union*, *Intersection*, *Difference*.
3. Create an ADT, *Bag*. In mathematics a *bag* is similar to a *set* except that a bag may contain duplicate elements. The minimal operations should include *Create*, *Insert*, *Remove*, and *IsIn*.

14 BASICS OF C++

The expression "a better C" is often used to describe the C++ language, implying that:

- (1) C and C++ have a lot of features in common.
- (2) C++ has a number of features not associated with either data abstraction or inheritance that improve on C.

In this section we will review features of the C++ language and identify

similarities and differences with C. Our presentation assumes that readers are familiar with the C language.

1.4.1 Program Organization in C++

As in C, a single C++ program is usually spread out over a number of files. Two kinds of files are used in C++: header files and source files. Header files have a `.h` suffix and are used to store declarations. Some header files (such as `<iostream>`) are system-defined. (The `.h` suffix is omitted for system-defined header files.) Others are user-defined. Source files are used to store C++ source code. The suffix used in source-file names depends on the compiler. We will use a `.C` suffix for source files. Header files are included in the appropriate files through the use of the `#include` preprocessor directive.

Before a C++ program can run, its source files must be individually compiled, linked, and loaded. Because a C++ program typically consists of many source and header files, it is possible that a header file gets included multiple times, giving rise to compilation errors. To ensure that a header file is never included twice, the contents of a header file are enclosed inside the following preprocessor directives.

```
#ifndef FILENAME_H
#define FILENAME_H
// insert contents of the header file here
-
-
-
#endif
```

Another consequence of using multiple source files is that the process of typing compilation commands every time we modify and recompile a program becomes rather tedious. To avoid this, we suggest that you use the *make* facility in UNIX, or its equivalent, to maintain a C++ program. In addition to saving the effort of repeatedly typing compilation commands, *make* also saves time by not recompiling source files that are not affected by modifications made to other files.

1.4.2 Scope in C++

Another consequence of using many files in a single program is that questions arise about the visibility of a variable declared in one file and used in other files. In view of this, we take a brief look at program scope in C++. C++ program text can be classified into one of four scopes:

14 Basic Concepts

File scope: Declarations that are not contained in a function definition, a class definition, or a namespace belongs to a file scope.

Namespace scope: A namespace is a mechanism that permits logically related variables and functions to be grouped together. For example, the standard library is defined in a namespace called `std`. In order to access an entity defined in a namespace from outside that namespace, it is necessary to provide the scope information. For example, the `cout` operator in the standard library is accessed using `std::cout`. The repeated use of the scope information may be avoided through the use of the `using` declaration (as in `using std`).

Local scope: A name declared in a block belongs to a local scope consisting of that block. (A block is a section of code delimited by a `{ }` pair.) It can also be used by sub-blocks contained within the block in which it was declared.

Class scope: Declarations associated with a class definition belongs to a class scope. Each class represents a distinct class scope. We discuss classes in greater detail later in Chapter 2.

Each variable has a scope or a context. A variable is uniquely identified by its scope and its name. A variable is visible to a program only from within its scope. For example, a variable defined in a block can only be accessed from within the block. A variable defined at file scope (a global variable), however, can be accessed anywhere in the program. Some questions that arise as a result are

(1) What do we do if a local variable reuses a global variable name in a block; but, we want to access the global variable?

Solution: use the scope operator `::` to access the global variable.

(2) A global variable is defined in `file1.C`, but used in `file2.C`. How do we declare the global variable in `file2.C`? If we declare it, as we normally declare variables, in `file2.C`, the compiler tells us that it is multiply defined. If we don't declare it, the compiler tells us that it hasn't been defined!

Solution: we extern to declare the variable in `file2.C`.

(3) In our program, both `file1.C` and `file2.C` define a global variable with the same name; but the two global variables are meant to be different entities. How do I get the program to compile without changing the global variable name in one of the files.

Solution: use `static` to declare the variables in both files.

For further details, the reader is referred to one of the introductory texts on C++

listed in the references at the end of the chapter.

1.4.3 C++ Statements and Operators

C++ statements have the same syntax and semantics as C statements. C++ statements are briefly reviewed later in this chapter in the context of performance analysis and evaluation.

C++ operators are identical to C operators with the exception of `new` and `delete`. We will study `new` and `delete` shortly when we discuss dynamic memory management in C++. Another difference is that C++ uses the shift left (`<<`) and shift right (`>>`) operators for input and output. We will also study these shortly when discussing input/output in C++. An important difference between C and C++ is that C++ allows *operator overloading*; that is, an operator is allowed to have different functions depending on the types of the operands that it is being applied to, and furthermore, these functions can be defined by a programmer. We will discuss operator overloading in detail in Chapter 2.

1.4.4 Data Declarations in C++

A data declaration associates a data type with a name. We have already listed the fundamental types of data provided by C++ in Section 1.3 on Data Abstraction and Encapsulation. Now, we list the various options available for declaring data in C++ below. All of these options are available in C with the exception of the reference type.

- (1) **Constant values:** these consist of literals such as 5, 'a', or 4.3.
- (2) **Variables:** these are instances of data types that can be modified during the course of a program.
- (3) **Constant variables:** these are variables that cannot be assigned a value during their lifetime. Because of this, they must be initialized; that is, their contents must be fixed at the time they are declared. A constant type is declared by adding the keyword `const` to its declaration (e.g., `const int MAX = 500` ;).
- (4) **Enumeration types:** This is an alternate mechanism for declaring a series of integer constants. It can also be used to create a new data type by giving the enumeration type a name:

```
enum seasoner { SUMMER, FALL, SPRING };
```


16 Basic Concepts

SUMMER, FALL and SPRING are constants of type *semester*. Their values are 0, 1 and 2, respectively, by default.

- (5) Pointers: these hold memory addresses of objects. For example,

```
int i = 25 ;  
int *np ;  
np = &i ;
```

Here *np* is declared to be a pointer to an integer. Next, it is assigned the address of integer variable *i*. So, *np* is now a pointer to the integer stored in variable *i* (in this case 25).

- (6) Reference types: This is a feature of C++ that is not a feature of C. The reference type is a mechanism to provide an alternate name for an object. A reference to an object of type *T* is declared by appending an *&* to *T* (i.e., *T&*). For example,

```
int i = 5 ;  
int& j = i ;  
i = 7 ;  
printf ("i= %d, j= %d", i, j) ;
```

In the example *j* is a reference type. It represents an alternate name for *i*. When *i*'s value is changed, *j*'s value also changes correspondingly. When the *printf* statement is executed, the final value of *i* and *j* is 7.

1.4.5 Comments in C++

Comments in C++ are specified in two ways:

- (1) Multiline comments: these are identical to comments in C. All text enclosed within the */* */* delimiters is ignored by the compiler.
- (2) Single line comments: these are unique to C++. All text after *//* on a line is ignored by the compiler.

The C++ single-line comment format overcomes the following disadvantage of the C multi-line comment: if one multi-line comment is nested inside another, the portion of the text between the terminating delimiters of the inner and outer comments is not ignored by the compiler. This could result in the inclusion of code

into the program that the programmer believes was commented out.

1.4.6 Input/Output in C++

To perform I/O in C++, it is necessary to include the system-defined header file *iostream*. The keyword *cout* is used for output to the standard output device. The *cout* keyword and each entity being printed are separated from each other by the *<<* operator. The entities being output are printed in left to right order on the standard output device.

```
#include <iostream>

main()
{
    int n = 50 ; float f = 20.3 ;
    cout << "n: " << n << endl ;
    cout << "f: " << f << endl ;
}
```

Program 1.1: Output in C++

Program 1.1 prints the following on the standard output device:

```
n: 50
f: 20.3
```

The *cin* keyword is used for input in C++. The operator *>>* is used to separate variables being input. Whitespace (i.e., the tab, newline, or blank characters) is used to separate items corresponding to different variables on the standard input device.

When Program 1.2 is executed, both of the following inputs on the standard input device result in variable *a* being set to 5 and *b* to 10:

Input1:
5 10 <Enter>

Input2:
5 <Enter>

18 Basic Concepts

```
#include <iostream>

main()
{
    int a, b ;
    cin >> a >> b ;
}
```

Program 1.3: Input in C++

```
10 <Enter>
```

An advantage of I/O in C++ is that it is format-free; that is, the programmer is not required to use formatting symbols to specify the type and order of items being input/output. Another advantage is that, like other C++ operators, I/O operators can be overloaded.

File I/O in C++ is performed by including the header file *fstream* as shown in Program 1.3. A *fstream* variable (*outFile*, in our example) is initialized with a string which represents the name of the file (here *my.out*) and a second argument which specifies the mode in which the file will be used (e.g., *ios::out* specifies that the file is to be used for output). If the file is not opened, *outFile* = 0. If *outFile* is successfully opened, it is used instead of *cout* to direct output to file *my.out*.

1.4.7 Functions in C++

There are two kinds of functions in C++: regular functions and member functions. Member functions are functions that are associated with specific C++ classes. We postpone a detailed discussion of member functions to Chapter 2. The features of functions that we review here are common to both types of functions.

A function consists of a function name, a list of arguments or *signature* (input), a return type (output), and the body (code that implements a function). In Program 1.4, *Max* is the function name, *int a*, *int b* is the list of arguments, *int* is the return type, and the statements between { and } form the body of the function.

All functions in C++ return a value. If a function is not meant to return anything, we use *void* to denote its return type. A value is returned from a function by

```
#include <iostream>
#include <fstream>

main()
{
    ofstream outFile("my.out", ios::out);
    if (!outFile) {
        cerr << "cannot open my.out " << endl; // standard error device
        return;
    }
    int n = 50 ; float f = 20.3 ;
    outFile << "n: " << n << endl;
    outFile << "f: " << f << endl;
}
```

Program 1.3: File I/O in C++

```
int Max (int a, int b)
{
    if (a > b) return a;
    else return b;
}
```

Program 1.4: An example of a function

using the return statement. The return statement must return a value that is of the same type as the function's return type or can be converted to the desired type. The function terminates when a return statement is encountered. A function is invoked by supplying the actual arguments (e.g., *Max*(*x*, 100) returns the larger of *x* and 100).

1.4.8 Parameter Passing in C++

In C++, arguments are passed by *value*. This is the default parameter-passing mechanism. When an object is passed by value, it is copied into the function's local storage. The function accesses this local copy. Consequently, any change

20 Basic Concepts

made to a parameter that is passed by value inside the function only changes its local copy. In other words, passing by value does not affect the actual arguments.

An argument may also be passed by reference. To do this, one needs to explicitly declare it to be a reference type. This is done by appending an `&` to its type specifier. In Program 1.4, we can pass `a` by reference by declaring it as `int& a` in the function's argument list. When an object is passed by reference, only the address of its location is copied into the function's local store; the object itself is not copied into the function's local store. Thus, the function accesses the objects referred to by the address; i.e., the actual arguments.

The advantage of passing by value is that variables that were supplied as actual arguments to the function are not inadvertently modified by it. The advantage of passing by reference is that the function executes faster if the object being passed requires more memory than its address. This is because the overhead of copying the actual argument into the function's local store is replaced by the overhead of copying its address. Note that pass-by-reference would be slower for an argument type such as `char` that occupies less memory than its address. Another reason for using reference parameters is if you want the function to modify the actual argument. This technique is used if a function is to return two or more items to the calling function. One of these items is returned by using the return statement. The others are returned through the use of reference arguments.

One technique for retaining the advantages of both parameter-passing methods is to pass *constant references* such as `const T& a`, where `T` is the type of the argument. Any attempt to modify a `const` argument in the function body will result in a compile-time error. It is also helpful to use the `const` keyword to characterize an argument that is not modified in a function, because it conveys this information to a user of the function. For a more detailed discussion on parameter passing mechanisms and the circumstances in which they should be used, see the book *Effective C++*, by Scott Meyers, listed in the references at the end of the chapter.

There is one exception to the rule that the default parameter passing mechanism in C++ is pass-by-value: array types are passed by reference; that is, the array is not copied into the function's local store. Therefore, any changes made to the array inside the function are reflected in the actual array. When a function is invoked with an array argument `a` (e.g., `f(a)`), a pointer to the first element of `a` (i.e., `&a[0]`) is actually being passed. This is the reason that, in function definitions, arrays are usually denoted by pointers to the appropriate type (e.g., `f(int *)`). Because arrays are passed in this manner, the function does not know the size of the array. This is typically rectified by explicitly passing the size of the array as a separate parameter of the function.

1.4.9 Function Name Overloading in C++

Finally, we discuss *function name overloading* in C++. Function overloading means that there can be more than one function with the same name *as long as they have different signatures*. For example, C++ would allow all the following to coexist even though they have the same name *Max*:

```
int Max(int, int);
int Max(int, int, int);
int Max(int*, int);
int Max(float, int);
int Max(int, float);
```

1.4.10 Inline Functions

An inline function is declared by adding the keyword *inline* to the function definition as follows:

```
inline int Sum( int a, int b)
{
    return a + b ;
}
```

The *inline* keyword tells the compiler that any calls to *Sum* must be replaced by the body of *Sum*. This eliminates the overhead of performing a function call and copying arguments when the program is executing. So, the statement *i = Sum (x, 12);* is replaced by *i = x + 12 ;*.

The objective of the *inline* and *const* keywords is to eliminate the use of preprocessor directives such as *#define*, which have traditionally been used to perform macro substitution. Excessive use of preprocessor directives makes it difficult to use programming tools such as debuggers and profilers effectively.

1.4.11 Dynamic Memory Allocation in C++

Objects may be allocated from and released to the free store during runtime through the use of the *new* and *delete* operators. The *new* operator creates an object of the desired type and returns a pointer to the data type that follows it. If it is unable to create the desired object, it throws an exception. An object created by *new* exists for the duration of the program unless it is explicitly deleted by applying the *delete* operator to a pointer to the object:

22 Basic Concepts

```
int *ip = new int;  
.  
.  
delete ip ;
```

Operators `new` and `delete` can also be used to create and delete an array of objects as follows:

```
int *ip = new int[10] ; // ip is an array of integers  
.  
.  
delete [] ip ; // delete the array ip
```

The operator `[]` is used to inform the compiler that the object being created or deleted is an array.

14.12 Exceptions

Throwing an Exception

Exceptions are used to signal the occurrence of errors and other special conditions. For example, the evaluation of the expression $a + b * c + b / c$ with $a = 2$, $b = 1$, and $c = 0$ requires us to divide by zero, which is an error. Although this error is not detected by the C++ compiler, your hardware will detect the error and throw an exception.

We can write C++ programs that check for exceptional conditions and throw an exception when such a condition is detected. For example, we may wish to write a function `DivZero` that computes the expression $a + b * c + b / c$ only when each of a , b , and c is greater than 0. Such a function would first check that the values of a , b , and c are actually > 0 . If one or more of these is ≤ 0 , we can signal an exceptional condition by throwing an exception as is done in Program 1.5. The exception thrown by this program is of type `char*` and the return expression is not evaluated.

We get more flexibility in processing exceptions when we define an exception class (or type) for each of the different kinds of exceptions (e.g., divide by zero, illegal parameter value, illegal input value, array index out of range) that our program may throw. For example, the C++ operator `new` that does dynamic memory allocation throws an exception of type `bad_alloc` when it is unable to make the requested memory allocation.

Handling Exceptions

Exceptions that might be thrown by a piece of code can be handled by enclosing this code within a `try` block. The `try` block is then followed by zero or more

```
int DivZero(int a, int b, int c)
{
    if (a <= 0 || b <= 0 || c <= 0)
        throw "All parameters should be > 0";
    return a + b * c + b / c;
}
```

Program 1.5: Throwing an exception of type `char*`

catch blocks. Each catch block has a parameter or argument whose type determines the type of exception that may be caught by that catch block. For example, the block

```
catch (char* e) {}
```

catches exceptions of type `char*` while the block

```
catch (bad_alloc e) {}
```

catches exceptions of type `bad_alloc`. The block

```
catch (...) {}
```

catches all exceptions regardless of their type.

A catch block typically contains code to recover from the exception that has occurred, or if recovery is not possible, the code in the catch block prints out an error message. Program 1.6 shows an example of the try-catch construct.

Although Program 1.6 has a single catch block following the try block, it is possible to follow a try block with several catch blocks. When the code within a try block terminates with no exception, we bypass the catch blocks. When an exception is thrown, normal execution of the try block terminates and we enter the first catch block that matches an exception of the type thrown. Following the execution of the code within this matching catch block, we bypass the remaining catch blocks. If no catch block matches the thrown exception type, then the exception propagates through the hierarchy of nested enclosing try blocks to the first catch block in this hierarchy that can handle the exception. If the exception is not caught by any catch block, the program terminates abnormally.

When Program 1.6 executes, `DivZero` throws an exception of type `char*`. This exception causes `DivZero` to terminate without the evaluation of the expression. Also, the try block terminates immediately (the `cout` in the try block

24 Basic Concepts

```
int main()
{
    try {cout << DivZero(2,0,4) << endl;}
    catch (char* e)
    {
        cout << "The parameters to DivZero were 2, 0, and 4" << endl;
        cout << "An exception has been thrown" << endl;
        cout << e << endl;
        return 1;
    }
    return 0;
}
```

Program 1.6: Catching an exception of type `char*`

doesn't complete). Since the type of the exception thrown by *DivZero* is the same as that of the catch block's parameter *e*, the exception is caught by this catch block; *e* is assigned the thrown exception; and the catch block is entered. Figure 1.2 gives the output generated by Program 1.6.

```
The parameters to DivZero were 2, 0, and 4
An exception has been thrown
All parameters should be > 0
```

Figure 1.2: Output from Program 1.6

EXERCISES

1. Modify Program 1.5 so that it throws an exception of type `int`. The value of the thrown exception should be 1 if *a*, *b*, and *c* are all less than 0; the value should be 2 if all three equal 0. When neither of these conditions is satisfied, no exception is thrown. Write a `main` function that uses your modified code, catches the exception if thrown, and outputs a message that depends on the value of the thrown exception. Test your code.
2. Write a C++ function to return the sum of the first *n* numbers in the integer array *a*. Your function should throw an exception in case *n* < 0. Test your code.

1.5 ALGORITHM SPECIFICATION

1.5.1 Introduction

The concept of an algorithm is fundamental to computer science. Algorithms exist for many common problems, and designing efficient algorithms plays a crucial role in developing large-scale computer systems. Therefore, before we proceed further, we discuss this concept more fully. We begin with a definition.

Definition: An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

- (1) **Input.** Zero or more quantities are externally supplied.
- (2) **Output.** At least one quantity is produced.
- (3) **Definiteness.** Each instruction is clear and unambiguous.
- (4) **Finiteness.** If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
- (5) **Effectiveness.** Every instruction must be basic enough to be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in (3); it also must be feasible. \square

In computational theory, one distinguishes between an algorithm and a program, the latter of which does not have to satisfy the fourth condition. For example, we can think of an operating system that continues in a “wait” loop until more jobs are entered. Such a program does not terminate unless the system crashes. Since our programs will always terminate, we will use the terms algorithm and program interchangeably in this text.

We can describe an algorithm in many ways. We can use a natural language like English, although if we select this option, we must make sure that the resulting instructions are definite. Graphic representations called flowcharts are another possibility, but they work well only if the algorithm is small and simple. In this text, we will present most of our algorithms in C++, occasionally resorting to a combination of English and C++ for our specifications. Two examples should help to illustrate the process of translating a problem into an algorithm.

Example 1.2 [Selection sort]: Suppose we must devise a program that sorts a collection of $n \geq 1$ integers. A simple solution is given by the following:

26 Basic Concepts

From those integers that are currently unsorted, find the smallest and place it next in the sorted list.

Although this statement adequately describes the sorting problem, it is not an algorithm because it leaves several unanswered questions. For example, it does not tell us where and how the integers are initially stored or where we should place the result. We assume that the integers are stored in an array, a , such that the i th integer is stored in $a[i-1]$, $1 \leq i \leq n$. Program 1.7 is our first attempt at deriving a solution. Notice that it is written partially in C++ and partially in English.

```
for (int i = 0; i < n; i++) {  
    examine  $a[i]$  to  $a[n-1]$  and suppose the smallest integer is at  $a[j]$ ;  
    interchange  $a[i]$  and  $a[j]$ ;  
}
```

Program 1.7: Selection sort algorithm

To turn Program 1.7 into a real C++ program, two clearly defined subtasks remain: finding the smallest integer and interchanging it with $a[i]$. We can solve the latter problem by using the code:

$$\text{temp} = a[i]; a[i] = a[j]; a[j] = \text{temp};$$

We implement this by calling the standard C++ function `swap(a[i], a[j])`, whose arguments are passed by reference. The first subtask can be solved by assuming the minimum is $a[i]$, checking $a[i]$ with $a[i+1]$, $a[i+2]$, \dots and whenever a smaller element is found, regarding it as the new minimum. Eventually $a[n-1]$ is compared to the current minimum, and we are done. Putting all these observations together, we get the function `sort` (Program 1.8).

At this point, the obvious question to ask is: Does this function work correctly?

Theorem 1.1: *SelectionSort(a, n) correctly sorts a set of $n \geq 1$ integers; the result remains in $a[0] \dots a[n-1]$ such that $a[0] \leq a[1] \leq \dots \leq a[n-1]$.*

Proof: We first note that for any i , say $i = q$, following the execution of lines 5–9, it is the case that $a[q] \leq a[r]$, $q < r \leq n-1$. Also observe that when i becomes greater than q , $a[0] \dots a[q]$ is unchanged. Hence, following the last execution of these lines (i.e., $i = n-1$), we have $a[0] \leq a[1] \leq \dots \leq a[n-1]$. \square

We observe at this point that the upper limit of the for loop in line 3 can be

```

1 void SelectionSort (int *a, const int n)
2 { // Sort the n integers a[0] to a[n-1] into nondecreasing order.
3   for (int i = 0; i < n; i++)
4   {
5     int j = i;
6     // find smallest integer in a[i] to a[n-1]
7     for (int k = i + 1; k < n; k++)
8       if (a[k] < a[j]) j = k;
9     swap (a[i], a[j]);
10  }
11 }

```

Program 1.8: Selection sort

changed to $n - 1$ without damaging the correctness of the algorithm.

Example 1.3 [Binary search]: Assume that we have $n \geq 1$ distinct integers that are already sorted and stored in the array $a[0] \dots a[n-1]$. Our task is to determine if the integer x is present and if so to return j such that $x = a[j]$; otherwise return -1 . By making use of the fact that the set is sorted, we conceive of the following efficient method:

Let *left* and *right*, respectively, denote the left and right ends of the list to be searched. Initially, *left* = 0 and *right* = $n-1$. Let *middle* = (*left*+*right*)/2 be the middle position in the list. If we compare $a[\text{middle}]$ with x , we obtain one of three results:

- (1) $x < a[\text{middle}]$. In this case, if x is present, it must be in the positions between 0 and *middle* - 1. Therefore, we set *right* to *middle* - 1.
- (2) $x == a[\text{middle}]$. In this case, we return *middle*.
- (3) $x > a[\text{middle}]$. In this case, if x is present, it must be in the positions between *middle* + 1 and $n-1$. So, we set *left* to *middle* + 1.

If x has not been found and there are still integers to check, we recalculate *middle* and continue the search. The algorithm contains two subtasks: (1) determining if there are any integers left to check and (2) comparing x to $a[\text{middle}]$.

At this point you might try the method out on some sample numbers. This method is referred to as *binary search*. Note how at each stage the number of elements in the remaining set is decreased by about one-half. Note also that at

28 Basic Concepts

each stage, x is compared with $a[\text{middle}]$ and depending on whether $x > a[\text{middle}]$, $x < a[\text{middle}]$, or $x = a[\text{middle}]$, we do a different thing. We can now refine the description of binary search to get a pseudo-C++ function. The result is given in Program 1.9.

```
int BinarySearch (int *a, const int x, const int n)
{ // Search the sorted array a[0], ..., a[n-1] for x
  Initialize left and right;
  while (there are more elements)
  {
    Let middle be the middle element;
    if (x < a[middle]) set right to middle-1;
    else if (x > a[middle]) set left to middle+1;
    else return middle;
  }
  Not found;
}
```

Program 1.9: Algorithm for binary search

Another refinement yields the C++ function of Program 1.10.

```
int BinarySearch (int *a, const int x, const int n)
{ // Search the sorted array a[0], ..., a[n-1] for x.
  int left = 0, right = n - 1;
  while (left <= right)
  { // there are more elements
    int middle = (left + right)/2;
    if (x < a[middle]) right = middle-1;
    else if (x > a[middle]) left = middle+1;
    else return middle;
  } // end of while
  return -1; // not found
}
```

Program 1.10: C++ function for binary search

To prove this program correct we make assertions about the relationship between variables before and after each iteration of the while loop. As we enter this loop

and if x is present in a , the following holds:

$$left \leq right \ \&\& \ a[left] \leq x \leq a[right] \ \&\& \ SORTED(a, n)$$

Now, if control passes out of the while loop, then we know the condition of while loop is false, so $left > right$. This, combined with the above assertion, implies that x is not present.

Unfortunately, a complete proof takes us beyond the scope of this text, but those who wish to pursue program-proving should consult the references at the end of this chapter. \square

1.5.2 Recursive Algorithms

We have emphasized the need to structure a program to make it easier to achieve the goals of readability and correctness. One of the most useful syntactical features for accomplishing this is the function. A set of instructions that perform a logical operation, perhaps a very complex and long operation, can be grouped together as a function. The function name and its parameters are viewed as a new instruction that can be used in other programs. Given the input-output specifications of a function, we do not even have to know how the task is accomplished, only that it is available. This view of the function implies that it is invoked, executed and returns control to the appropriate place in the calling function. What this fails to stress is that functions may call themselves (*direct recursion*) before they are done or they may call other functions that again invoke the calling function (*indirect recursion*). These recursive mechanisms are extremely powerful, but even more importantly, often they can express an otherwise complex process very clearly. For these reasons we introduce recursion here.

Recursion is similar to the method of induction which is often used to prove mathematical statements. In mathematical induction, a statement about integers (e.g., the sum of the first n positive integers is $n(n+1)/2$) is proved by showing that the statement can be proved for integer k if it is assumed to be true for integer $k-1$. Similarly, in recursion, we write a function to produce an output (say $n!$) for some input (here, n) by assuming that the same function will compute the correct output for input $n-1$. In mathematical induction, we need a basis which can be directly proved (that is, the proof for the basis does not make any assumptions). Similarly, a recursive function requires a terminating condition. When the input to the function satisfies this terminating condition, the function directly computes the output without calling itself.

What kinds of problems are best solved by recursion? Typically, beginning programmers view recursion as a somewhat mystical technique that is useful only for some very special class of problems (such as computing factorials or

30 Basic Concepts

Ackermann's function). This is unfortunate because any program that can be written using assignment, the if-else statement, and the while statement can also be written using assignment, if-else, and recursion. Of course, this does not mean that the resulting program will necessarily be easier to understand. However, there are many instances when this will be the case. When is recursion an appropriate mechanism for algorithm exposition? One instance is when the problem itself is recursively defined. Factorial fits this category, as well as binomial coefficients where

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}$$

can be recursively computed by the formula

$$\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$$

We use two examples to show you how to develop a recursive algorithm. In the first example, we take the binary search function that we created in Example 1.3 and transform it into a recursive function. In the second example, we generate all possible permutations of a list of characters. To understand a recursive function, you must

- (1) Formulate in your mind a statement of what it is that the function is supposed to do, for a given input.
- (2) Verify that the function does achieve its goal if the recursive invocations to itself do what they are supposed to.
- (3) Ensure that a finite number of recursive invocations of the function eventually lead to an invocation which satisfies the terminating condition (otherwise, the function will keep calling itself and not terminate!).
- (4) The function should perform the correct computations if the terminating condition is encountered.

Example 1.4 [Recursive binary search]: Program 1.10 gave the iterative version of binary search. In the recursive version we pass *left* and *right* as parameters (Program 1.11). The *for* loop of Program 1.10 has been replaced by recursive calls in Program 1.11. To invoke the recursive function, we use the statement

BinarySearch(a, x, 0, n-1);

You should verify that *BinarySearch* satisfies the four conditions stated above for recursive functions. Notice that both the iterative (Program 1.10) and recursive

(Program 1.11) functions perform the same computation. \square

```

int BinarySearch (int *a, const int x, const int left, const int right)
{ // Search the sorted array a[left], ..., a[right] for x
  if (left <= right) {
    int middle = (left + right)/2;
    if (x < a[middle]) return BinarySearch(a, x, left, middle - 1);
    else if (x > a[middle]) return BinarySearch(a, x, middle + 1, right);
    return middle;
  } // end of if
  return -1; // not found
}

```

Program 1.11: Recursive implementation of binary search

Example 1.5 [Permutation generator]: Given a set of $n \geq 1$ elements, the problem is to print all possible permutations of this set. For example if the set is $\{a, b, c\}$, then the set of permutations is $\{(a, b, c), (a, c, b), (b, a, c), (b, c, a), (c, a, b), (c, b, a)\}$. It is easy to see that given n elements, there are $n!$ different permutations. A simple algorithm can be obtained by looking at the the case of four elements $\{a, b, c, d\}$. The answer can be constructed by writing

- (1) a followed by all permutations of $\{b, c, d\}$
- (2) b followed by all permutations of $\{a, c, d\}$
- (3) c followed by all permutations of $\{a, b, d\}$
- (4) d followed by all permutations of $\{a, b, c\}$

The expression “followed by all permutations” is the clue to recursion. It implies that we can solve the problem for a set with n elements if we have an algorithm that works on $n - 1$ elements. These observations lead to Program 1.12, which is invoked by `Permutations(a, 0, n - 1)`.

Try this algorithm out on sets of length one, two, and three to ensure that you understand how it works. \square

Another time when recursion is useful is when the data structure that the algorithm is to operate on is recursively defined. We shall see several important examples of such structures in this book.

```

void Permutations (char *a, const int k, const int m)
{ // Generate all the permutations of a[k], ..., a[m].
  if (k == m) { // output permutation
    for (int i = 0; i <= m; i++) cout << a[i] << " ";
    cout << endl;
  }
  else if a[k:m] has more than one permutation. Generate these recursively.
    for (i = k; i <= m; i++) {
      swap(a[k], a[i]);
      Permutations(a, k + 1, m);
      swap(a[k], a[i]);
    }
}

```

Program 1.12: Recursive permutation generator

EXERCISES

1. Horner's rule is a means for evaluating a polynomial $A(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$ at a point x_0 using a minimum number of multiplications. This rule is:

$$A(x) = (\cdots (a_n x_0 + a_{n-1}) x_0 + \cdots + a_1) x_0 + a_0$$

Write a C++ program to evaluate a polynomial using Horner's rule. Determine how many times each statement is executed.

2. Given n Boolean variables x_1, \cdots, x_n we wish to print all possible combinations of truth values they can assume. For instance, if $n = 2$, there are four possibilities: true, true; true, false; false, true; false, false. Write a C++ program to accomplish this and do a frequency count.
3. Trace the action of the code

```

i = 0; j = n - 1;
do {
  k = (i + j) / 2;
  if (a[k] <= x) i = k + 1;
  else j = k - 1;
} while (i <= j);

```

on the elements 2, 4, 6, 8, 10, 12, 14, 16, 18, and 20 searching for $x = 1, 3, 13$, or 21.

4. Write a C++ program that prints out the integer values of x , y , and z in non-decreasing order. What is the competing time of your method?
5. Write a C++ function that searches an unsorted array $a[0:n-1]$ for the element x . If x occurs, then return the leftmost position of x in the array, else return -1 .
6. The factorial function $n!$ has value 1 when $n \leq 1$ and value $n \cdot (n-1)!$ when $n > 1$. Write both a recursive and an iterative C++ function to compute $n!$.
7. The Fibonacci numbers are defined as: $f_0 = 0$, $f_1 = 1$, and $f_i = f_{i-1} + f_{i-2}$ for $i > 1$. Write both a recursive and an iterative C++ function to compute f_i .
8. Write a recursive function for computing the binomial coefficient $\binom{n}{m}$ as defined in Section 1.5.2, where $\binom{n}{0} = \binom{n}{n} = 1$. Analyze the time and space requirements of your algorithm.
9. Write an iterative function to compute a binomial coefficient; then transform it into an equivalent recursive function.
10. Ackermann's function $A(m, n)$ is defined as follows:

$$A(m, n) = \begin{cases} n + 1 & , \text{ if } m = 0 \\ A(m - 1, 1) & , \text{ if } n = 0 \\ A(m - 1, A(m, n - 1)) & , \text{ otherwise} \end{cases}$$

This function is studied because it grows very fast for small values of m and n . Write a recursive function for computing this function. Then write a nonrecursive algorithm for computing Ackermann's function.

11. The *pigeonhole principle* states that if a function f has n distinct inputs but less than n distinct outputs, then there exist two inputs a and b such that $a \neq b$ and $f(a) = f(b)$. Write a program to find the values a and b for which the range values are equal. Assume that the inputs are $1, 2, \dots, n$.
12. Given n , a positive integer, determine if n is the sum of all of its divisors — i.e., if n is the sum of all t such that $1 \leq t < n$, and t divides n .
13. Consider the function $F(x)$ defined by

$$\begin{aligned} \text{if } (x \text{ is even}) \quad & F = x / 2; \\ \text{else} \quad & F = F(F(3x + 1)); \end{aligned}$$

Prove that $F(x)$ terminates for all integers x . (Hint: Consider integers of the form $(2i + 1)2^b - 1$ and use induction.)

14. If S is a set of n elements, the *power set* of S is the set of all possible subsets of S . For example, if $S = \{a, b, c\}$, then $\text{powerset}(S) = \{\langle \rangle, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$. Write a recursive function to compute

34 Basic Concepts

powerset (S).

15. [Towers of Hanoi] There are three towers and sixty-four disks of different diameters placed on the first tower. The disks are in order of decreasing diameter as one scans up the tower. Monks were supposed to move the disks from tower 1 to tower 3 obeying the following rules: (a) only one disk can be moved at any time and (b) no disk can be placed on top of a disk with smaller diameter. Write a recursive function that prints the sequence of moves that accomplish this task.

1.6 THE STANDARD TEMPLATE LIBRARY

The C++ standard templates library (STL) is a collection of containers, adaptors, iterators, function objects (also known as functors) and algorithms. Through the judicious use of elements of the STL, the task of writing application codes is greatly simplified. In this section, we introduce a few of these elements. To use these STL elements in your programs, you should add the statement

```
#include <algorithm>
```

to your programs.

Example 1.6 [The STL algorithm *accumulate*]: The STL has an algorithm *accumulate* that may be used to sum the elements in a sequence. The syntax is

```
accumulate (start, end, initialValue)
```

where *start* points to the first element to be accumulated and *end* points to one position after the last element to be accumulated. So, elements in the range (*start*, *end*) are accumulated. The invocation

```
accumulate (a, a + n, initialValue)
```

where *a* is a one-dimensional array, for example, returns the value

$$\text{initialValue} + \sum_{i=0}^{n-1} a[i]$$

The STL algorithm *accumulate* accesses successive elements of the sequence to be summed by performing the ++ operator on *start* and terminating when the pointer value becomes *end*. So, this algorithm may be used to sum the values of any sequence whose elements may be obtained by repeated application

of the `++` operator. One-dimensional arrays and the STL container `vector` are two examples of sequences whose elements may be accessed in this way. We shall see other examples later in this book.

The STL has a more general form of the *accumulate* algorithm, which has the following syntax:

`accumulate(start, end, initialValue, operator)`

where *operator* is a function that defines the operation to be used during the accumulation process. Using the STL functor `multiples<int>`, which multiplies two integers, we can find the product of an array of integers using the code of Program 1.13. □

```
int Product(int *a, int n)
// Return sum of the numbers a[0:n-1].
{
    int initVal = 1;
    return accumulate(a, a + n, initVal, multiples<int>());
}
```

Program 1.13: Compute the product of the elements `a[0:n-1]`

Example 1.7: (The STL algorithms `copy` and `next_permutation`) The `copy` algorithm copies a sequence of elements from one location to another. The syntax is

`copy(start, end, to)`

where *to* gives the location to which the first element is to be copied. So, elements are copied from locations `start`, `start + 1`, ..., `end - 1` to the locations `to`, `to + 1`, ..., `to + end - start`.

The algorithm `next_permutation`, which has the syntax

`next_permutation(start, end)`

creates the next lexicographically larger permutation of the elements in the range `[start, end)`; it returns the value `true` if-and-only-if (iff) such a next permutation exists. By starting with the smallest lexicographic permutation of a sequence of distinct elements and making successive calls to `next_permutation`, we can obtain all permutations. Program 1.14 does just this. The invocation of `copy` in this program copies the elements `a[0:n]` to the output stream `cout`; each copied element is followed by a space (" "). Notice that Program 1.14 outputs no

36 Basic Concepts

permutation that is lexicographically smaller than the initial sequence whereas Program 1.15 outputs all permutations regardless of the initial sequence. The exercises examine ways to modify Program 1.14 so as to obtain all permutations.

```
void Permutations(char *a, const int n)
// Generate all permutations of a[0:n].
// Output the permutations one by one
do {
    copy(a, a + n + 1, ostream_iterator<char>(cout, " "));
    cout << endl;
} while (next_permutation(a, a + n + 1));
}
```

Program 1.14: Permutations using the STL algorithm *next_permutation*

A more general form of the *next_permutation* algorithm takes a third parameter *compare* as in

next_permutation(*start*, *end*, *compare*)

When this form is used, the binary function *compare* is used to determine whether one element is smaller than another. In the two-parameter version, this comparison is done using the operator *<*. □

The STL contains many algorithms in addition to the ones used in the preceding two examples. The exercises explore STL algorithms further.

EXERCISES

1. Modify Program 1.14 so that it outputs all permutations of distinct elements. Do this by sorting the element list into ascending order prior to generating the permutations. To sort, use the STL algorithm

sort(*start*, *end*)

which sorts elements in the range [*start*, *end*) into ascending order. Test your code.

2. Modify Program 1.14 so that it outputs all permutations of distinct elements. Do this by first using *next_permutation* to generate permutations that are lexicographically larger than the initial permutation and then using the STL

algorithm `prev_permutation` to generate permutations that are lexically smaller than the initial permutation. Test your code.

3. Modify Program 1.14 so that it outputs all permutations of distinct elements. Do this by using the fact that when `next_permutation` returns the value `false` the sequence `[start, end)` is the lexically smallest sequence. Hence, subsequent invocations of `next_permutation` will get you the remaining (if any) permutations you need. Test your code.
4. The STL algorithm `count`, which has the syntax

```
count(start, end, value)
```

returns the number of occurrences of `value` in the range `[start, end)`. Write a program that uses this algorithm to determine the number of occurrences of `a[0]` in the integer array `a[0:n-1]`. Test your code.

5. The STL algorithm `fill` which has the syntax

```
fill(start, end, value)
```

sets all positions in the range `[start, end)` to `value`. Write a program that creates an integer array of a specified size and initializes all positions of this array to 0. Test your code.

1.7 PERFORMANCE ANALYSIS AND MEASUREMENT

One goal of this book is to develop skills for making evaluative judgments about programs. There are many criteria upon which we can judge a program, for instance:

- (1) Does it do what we want it to do?
- (2) Does it work correctly according to the original specifications of the task?
- (3) Is there documentation that describes how to use it and how it works?
- (4) Are functions created in such a way that they perform logical subfunctions?
- (5) Is the code readable?

The above criteria are all vitally important when it comes to writing software, most especially for large systems. Though we will not be discussing how to reach these goals, we will try to achieve them throughout this book with the programs we write. Hopefully this more subtle approach will gradually infect your

38 Basic Concepts

own program-writing habits so that you will automatically strive to achieve these goals.

There are other criteria for judging programs that have a more direct relationship to performance. These have to do with their computing time and storage requirements.

Definition: The *space complexity* of a program is the amount of memory it needs to run to completion. The *time complexity* of a program is the amount of computer time it needs to run to completion. \square

Performance evaluation can be loosely divided into two major phases: (1) *a priori* estimates and (2) *a posteriori* testing. We refer to these as *performance analysis* and *performance measurement* respectively.

1.7.1 Performance Analysis

1.7.1.1 Space Complexity

Function *Abr* (Program 1.16) computes the expression $a+b+b*c+(x+b-c)/(a+b)+4.0$; function *Sum* (Program 1.17) computes the sum $\sum_{i=0}^{n-1} a[i]$ iteratively, where the $a[i]$'s are of type `float`; and function *Rsum* (Program 1.18) is a recursive program that computes $\sum_{i=0}^{n-1} a[i]$.

```
float Abr(float a, float b, float c)
{
    return a+b+b*c+(a+b-c)/(a+b)+4.0;
}
```

Program 1.16: Function to compute $a+b+b*c+(x+b-c)/(a+b)+4.0$

The space needed by each of these programs is seen to be the sum of the following components:

- (1) A fixed part that is independent of the characteristics (e.g., number, size) of the inputs and outputs. This part typically includes the instruction space (i.e., space for the code), space for simple variables and fixed-size component variables (also called aggregate), space for constants, etc.

```

line float Sum (float *a, const int n)
1 {
2   float s = 0;
3   for (int i = 0; i < n; i++)
4     s += a[i];
5   return s;
6 }

```

Program 1.17: Iterative function for sum

```

line float Rsum (float *a, const int n)
1 {
2   if (n <= 0) return 0;
3   else return (Rsum(a, n-1) + a[n-1]);
4 }

```

Program 1.18: Recursive function for sum

- (2) A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables (to the extent that this depends on instance characteristics), and the recursion stack space (insofar as this space depends on the instance characteristics).

The space requirement $S(P)$ of any program P may therefore be written as $S(P) = c + S_P(\text{instance characteristics})$ where c is a constant.

When analyzing the space complexity of a program, we shall concentrate solely on estimating S_P (instance characteristics). For any given problem, we shall need to first determine which instance characteristics to use to measure the space requirements. This is very problem-specific, and we shall resort to examples to illustrate the various possibilities. Generally speaking, our choices are limited to quantities related to the number and magnitude of the inputs to and outputs from the program. At times, more complex measures of the interrelationships among the data items are used.

Example 1.8: For Program 1.16, the problem instance is characterized by the specific values of a , b , and c . Making the assumption that one word is adequate

40 Basic Concepts

to store the values of each of a , b , c , and the value returned by Abc , we see that the space needed by function Abc is independent of the instance characteristics. Consequently, $S_P(\text{instance characteristics}) = 0$. \square

Example 1.9: The problem instances for Program 1.17 are characterized by n , the number of elements to be summed. Since n is passed by value, one word must be allocated for it. Since a is actually the address of the first element in $a[1]$ (i.e. $a[0]$), the space needed by it is also one word. So, the space needed by the function is independent of n and $S_{\text{sum}}(n) = 0$. \square

Example 1.10: Let us consider the function $Rsum$. As in the case of Sum , the instances are characterized by n . The recursion stack space includes space for the formal parameters, the local variables, and the return address. Since a is the address of $a[0]$, it requires only one word of memory on the stack. Assume that the return address requires only one word of memory. Each call to $Rsum$ requires at least 4 words (including space for the values of n , a , the returned value, and the return address). Since the depth of recursion is $n + 1$, the recursion stack space needed is $4(n + 1)$. For $n = 1000$, the recursion stack space is 4004. \square

1.7.1.2 Time Complexity

The time, $T(P)$, taken by a program P is the sum of the compile time and the run (or execution) time. The compile time does not depend on the instance characteristics. Also, we may assume that a compiled program will be run several times without recompilation. Consequently, we shall concern ourselves with just the run time of a program. This run time is denoted by t_P (instance characteristics).

Because many of the factors t_P depends on are not known at the time a program is conceived, it is reasonable to attempt only to estimate t_P . If we knew the characteristics of the compiler to be used, we could proceed to determine the number of additions, subtractions, multiplications, divisions, compares, loads, stores, and so on that would be made by the code for P . So, we could obtain an expression for $t_P(n)$ of the form

$$t_P(n) = c_a \text{ADD}(n) + c_s \text{SUB}(n) + c_m \text{MUL}(n) + c_d \text{DIV}(n) + \dots$$

where n denotes the instance characteristics, and c_a , c_s , c_m , c_d , etc., respectively, denote the time needed for an addition, subtraction, multiplication, division, etc., and ADD , SUB , MUL , DIV , etc., are functions whose value is the number of additions, subtractions, multiplications, divisions, etc., that will be performed when the code for P is used on an instance with characteristic n .

Obtaining such an exact formula is in itself an impossible task, since the

time needed for an addition, subtraction, multiplication, etc., often depends on the actual numbers being added, subtracted, multiplied, etc. In reality then, the true value of $t_p(n)$ for any given n can be obtained only experimentally. The program is typed, compiled, and run on a particular machine. The execution time is physically clocked and $t_p(n)$ obtained. Even with this experimental approach, one could face difficulties. In a multiuser system, the execution time will depend on such factors as system load, the number of other programs running on the computer at the time program P is run, the characteristics of these other programs, and so on.

Given the minimal utility of determining the exact number of additions, subtractions, etc., that are needed to solve a problem instance with characteristics given by n , we might as well lump all the operations together (provided that the time required by each is relatively independent of the instance characteristics) and obtain a count for the total number of operations. We can go one step further and count only the number of program steps.

A *program step* is loosely defined as a syntactically or semantically meaningful segment of a program that has an execution time that is independent of the instance characteristics. For example, the entire statement

$$\text{return } a + b + b * c + (a + b - c) / (a + b) + 4.0;$$

of Program 1.16 could be regarded as a step since its execution time is independent of the instance characteristics (this statement is not strictly true, since the time for a multiply and divide will generally depend on the actual numbers involved in the operation).

The number of steps any program statement is to be assigned depends on the nature of that statement. The following discussion considers the various statement types that can appear in a C++ program and states the complexity of each in terms of the number of steps:

- (1) *Comments.* Comments are not executable statements and have a step count of zero.
- (2) *Declarative statements.* This category includes all statements that define or characterize variables and constants (int, long, short, char, float, double, const, enum, signed, unsigned, static, extern), all statements that enable users to define their own data types (class, struct, union, template), all statements that determine access (private, public, protected, friend), and all statements that characterize functions (void, virtual). These count as zero steps, since these are either not executable or their cost may be lumped into the cost of invoking the function they are associated with.
- (3) *Expressions and assignment statements.* Most expressions have a step count of one. The exceptions are expressions that contain function calls,

42 Basic Concepts

In this case, we need to determine the cost of invoking the functions. This cost can be large if the functions employ many-element pass-by-value parameters because the values of all actual parameters need to be assigned to the formal parameters. This is discussed further under function and function invocation. When the expression contains functions, the step count is the sum of the step counts assignable to each function invocation.

The assignment statement `<variable> = <expr>` has a step count equal to that of `<expr>` unless the size of `<variable>` is a function of the instance characteristics. In this latter case, the step count is the size of `<variable>` plus the step count of `<expr>`. For example, the assignment `a = b`, where `a` and `b` are of type *ElementList*, has a step count equal to the size of *ElementList*.

- (4) *Iteration statements.* This class of statements includes the **for**, **while**, and **do** statements. We shall consider the step counts only for the control part of these statements. These have the following form:

```
for (<init-stmt>; <expr1>; <expr2>)
while <expr> do
do ...while <expr>
```

Each execution of the control part of a **while** and **do** statement will be given a step count equal to the number of step counts assignable to `<expr>`. The step count for each execution of the control part of a **for** statement is one, unless the counts attributable to `<init-stmt>`, `<expr1>`, or `<expr2>` are a function of the instance characteristics. In this latter case, the first execution of the control part of the **for** has a step count equal to the sum of the counts for `<init-stmt>` and `<expr1>`; subsequent executions of the **for** statement have a step count equal to the sum of the step counts for `<expr1>` and `<expr2>`.

- (5) *Switch statement.* This statement consists of a header followed by one or more sets of condition-statement pairs. We shall once again consider the costs for the control part of the statement.

```
switch (<expr>) {
  case cond1: <statement1>
  case cond2: <statement2>
  .
  .
  .
  default: <statement>
}
```

The header switch (*<expr>*) is given a cost equal to that assignable to *<expr>*. The cost of each condition is its cost plus that of all preceding conditions.

- (6) *If-else statement.* The if-else statement consists of three parts:

```
if (<expr>) <statements1>;  
else <statements2>;
```

Each part is assigned the number of steps corresponding to *<expr>*, *<statements1>*, and *<statements2>* respectively. Note that if the else clause is absent, then no cost is assigned to it. Step counts are computed for the arithmetic-if operation in a similar manner.

- (7) *Function invocation.* All invocations of functions count as one step unless the invocation involves pass-by-value parameters whose size depends on the instance characteristics. In this latter case, the count is the sum of the sizes of these value parameters. If the function being invoked is recursive, then we must also consider the local variables in the function being invoked. The sizes of local variables that are characteristic-dependent need to be added to the step count.
- (8) *Memory management statements.* These include *new object*, *delete object*, and *sizeof(object)*. The step count associated with each is 1. The statements *new* and *delete* can potentially invoke the constructor and destructor for *object*, respectively. In this case, the step counts are computed in a similar manner to a function invocation.
- (9) *Function statements.* These count as zero steps because their cost has already been assigned to the invoking statements.
- (10) *Jump statements.* These include *continue*, *break*, *goto*, *return*, and *return <expr>*. Each has a step count of one, with the possible exception of *return <expr>*. This has a step count of 1 unless the step count attributable to *<expr>* is a function of instance characteristics. In this case, the step count is the cost of *<expr>*.

With the above assignment of step counts to statements, we can proceed to determine the number of steps needed by a program to solve a particular problem instance. We can go about this in one of two ways. In the first method, we introduce a new variable, *count*, into the program. This is a global variable with initial value 0. Statements to increment *count* by the appropriate amount are introduced into the program. This is done so that each time a statement in the original program is executed, *count* is incremented by the step count of that statement.

Example 1.11: When the statements to increment *count* are introduced into Program 1.17, the result is Program 1.19. The change in the value of *count* by the

44 Basic Concepts

time this program terminates is the number of steps executed by Program 1.17.

Since we are interested in determining only the change in the value of *count*, Program 1.19 may be simplified to Program 1.20. It should be easy to see that for every initial value of *count*, Program 1.19 and Program 1.20 compute the same final value for *count*. It is easy to see that in the *for* loop, the value of *count* will increase by a total of $2n$. If *count* is zero to start with, then it will be $2n+3$ on termination. So, each invocation of *Sum* (Program 1.17) executes a total of $2n+3$ steps. \square

```
float Sum (float *a, const int n)
{
    float s = 0;
    count++; // count is global
    for (int i = 0; i < n; i++) {
        count++; // for for
        s += a[i];
        count++; // for assignment
    }
    count++; // for last time of for
    count++; // for return
    return s;
}
```

Program 1.19: Program 1.17 with count statements added

```
void Sum (float *a, const int n)
{
    for (int i = 0; i < n; i++)
        count += 2;
    count += 3;
}
```

Program 1.20: Simplified version of Program 1.19

Example 1.12: When the statements to increment *count* are introduced into Program 1.18, Program 1.21 is obtained. Let $t_{\text{Sum}}(n)$ be the increase in the value of *count* when Program 1.21 terminates. We see that $t_{\text{Sum}}(0) = 2$. When $n > 0$, *count* increases by 2 plus whatever increase results from the invocation of *Sum*

from within the else clause. From the definition of t_{Rsum} , it follows that this additional increase is $t_{Rsum}(n-1)$. So, if the value of *count* is zero initially, its value at the time of termination is $2+t_{Rsum}(n-1)$, $n > 0$.

```

float Rsum (float *a, const int n)
{
    count++; // for if conditional
    if (n <= 0) {
        count++; // for return
        return 0;
    }
    else {
        count++; // for return
        return (Rsum(a, n - 1) + a[n - 1]);
    }
}

```

Program 1.21: Program 1.18 with count statements added

When analysing a recursive program for its step count, we often obtain a recursive formula for the step count (i.e., say $t_{Rsum}(n) = 2+t_{Rsum}(n-1)$, $n > 0$ and $t_{Rsum}(0) = 2$). These recursive formulas are referred to as *recurrence relations*. This recurrence may be solved by repeatedly substituting for t_{Rsum} as below:

$$\begin{aligned}
 t_{Rsum}(n) &= 2 + t_{Rsum}(n-1) \\
 &= 2 + 2 + t_{Rsum}(n-2) \\
 &= 2 \times 2 + t_{Rsum}(n-2) \\
 &\vdots \\
 &= 2n + t_{Rsum}(0) \\
 &= 2n + 2
 \end{aligned}$$

So, the step count for function *Rsum* (Program 1.18) is $2n + 2$. □

Comparing the step count of Program 1.17 to that of Program 1.18, we see that the count for Program 1.18 is less than that for Program 1.17. From this, we cannot conclude that Program 1.17 is slower than Program 1.18. This is so because a *step* does not correspond to a definite time unit. Each step of *Rsum* may take more time than every step of *Sum*. So, it might well be (and we expect)

46 Basic Concepts

that *Run* is slower than *Sum*.

The step count is useful in that it tells us how the run time for a program changes with changes in the instance characteristics. From the step count for *Sum*, we see that if n is doubled, the run time will also double (approximately); if n increases by a factor of 10, we expect the run time to increase by a factor of 10; and so on. So, we expect the run time to grow *linearly* in n .

Example 1.13 [Matrix addition]: Program 1.22 is a program to add two $m \times n$ matrices a and b together. Note that the argument `**a` refers to a two-dimensional array `a[][]`. Introducing the *count-incrementing* statements leads to Program 1.23. Program 1.24 is a simplified version of Program 1.23 that computes the same value for *count*. Examining Program 1.24, we see that line 5 is executed n times for each value of i or a total of mn times; line 6 is executed m times; and line 8 is executed once. If *count* is zero to begin with, it will be $2mn + 2n + 1$ when Program 1.24 terminates.

From this analysis we see that if $m > n$, then it is better to interchange the two *for* statements in Program 1.22. If this is done, the step count becomes $2mn + 2n + 1$. Note that in this example the instance characteristics are given by m and n . \square

```
line void Add (int **a, int **b, int **c, int m, int n)
1 {
2     for (int i = 0; i < m; i++)
3         for (int j = 0; j < n; j++)
4             c[i][j] = a[i][j] + b[i][j];
5 }
```

Program 1.22: Matrix addition

The second method to determine the step count of a program is to build a table in which we list the total number of steps contributed by each statement. This figure is often arrived at by first determining the number of steps per execution of the statement and the total number of times (i.e., frequency) each statement is executed. By combining these two quantities, the total contribution of each statement is obtained. By adding up the contributions of all statements, the step count for the entire program is obtained.

There is an important difference between the *step count* of a statement and its *steps per execution* (s/e). The step count does not necessarily reflect the complexity of the statement. For example, the statement

$$x = \text{Sum}(a, m);$$

```

line void Add (int **a, int **b, int **c, int m, int n)
{
    for (int i = 0; i < m; i++)
    {
        count++; // for for i
        for (int j = 0; j < n; j++)
        {
            count++; // for for j
            c[i][j] = a[i][j] + b[i][j];
            count++; // for assignment
        }
        count++; // for last time of for j
    }
    count++; // for last time of for i
}

```

Program 1.33: Matrix addition with counting statements

```

line void Add (int **a, int **b, int **c, int m, int n)
1      {
2          for (int i = 0; i < m; i++)
3          {
4              for (int j = 0; j < n; j++)
5                  count += 2;
6              count += 2;
7          }
8          count++;
9      }

```

Program 1.34: Simplified program with counting only

has a step count of 1, while the total change in *count* resulting from the execution of this statement is actually 1 plus the change resulting from the invocation of *Sum* (i.e., $2m+3$). The steps per execution of the above statement is $1+2m+3=2m+4$. The *slp* of a statement is the amount by which *count* changes as a result of the execution of that statement.

In Table 1.1, the number of steps per execution and the frequency of each of the statements in function *Sum* (Program 1.17) have been listed. The total

48 Basic Concepts

number of steps required by the program is determined to be $2n+3$. It is important to note that the frequency of line 3 is $n+1$ and not n . This is so because i has to be incremented to n before the for loop can terminate.

line	s/e	frequency	total steps
1	0	1	0
2	1	1	1
3	1	$n+1$	$n+1$
4	1	n	n
5	1	1	1
6	0	1	0
Total number of steps			$2n+3$

Table 1.1: Step table for Program 1.17

Table 1.2 gives the step count for function *Rsum* (Program 1.18). Line 2(a) refers to the if conditional of line 2, and line 2(b) refers to the statement in the if clause. Notice that under the s/e (steps per execution) column, line 3 has been given a count of $1+t_{Rsum}(n-1)$. This is the total cost of line 3 each time it is executed. It includes all the steps that get executed as a result of the invocation of *Rsum* from line 3. The frequency and total steps columns have been split into two parts: one for the case $n=0$ and the other for the case $n>0$. This is necessary because the frequency (and hence total steps) for some statements is different for each of these cases.

line	s/e	frequency		total steps	
		$n=0$	$n>0$	$n=0$	$n>0$
1	0	1	1	0	0
2(a)	1	1	1	1	1
2(b)	1	1	0	1	0
3	$1+t_{Rsum}(n-1)$	0	1	0	$1+t_{Rsum}(n-1)$
4	0	1	1	0	0
Total number of steps				2	$2+t_{Rsum}(n-1)$

Table 1.2: Step table for Program 1.18

Table 1.3 corresponds to function *Add* (Program 1.22). Once again, note

that the frequency of line 2 is $m+1$ and not m . This is so as i needs to be incremented up to m before the loop can terminate. Similarly, the frequency for line 3 is $m(m+1)$. When you have obtained sufficient experience in computing step counts, you may avoid constructing the frequency table and obtain the step count as in the following example.

line	s/c	frequency	total steps
1	0	1	0
2	1	$m+1$	$m+1$
3	1	$m(m+1)$	m^2+m
4	1	m	m
5	0	1	0
Total number of steps			$2m^2+2m+1$

Table 1.3: Step table for Program 1.22

Example 1.14 [Fibonacci numbers]: The Fibonacci sequence of numbers starts as

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \dots$$

Each new term is obtained by taking the sum of the two previous terms. If we call the first term of the sequence F_0 then $F_0 = 0$, $F_1 = 1$, and in general

$$F_n = F_{n-1} + F_{n-2}, \quad n \geq 2.$$

The program *Fibonacci* (Program 1.25) inputs any nonnegative integer n and prints the value F_n .

To analyze the time complexity of this program, we need to consider the two cases: (1) $n = 0$ or 1 and (2) $n > 1$. Line 3 will be regarded as two lines: 3(a), the conditional part, and 3(b), the if clause. When $n = 0$ or 1, lines 3(a) and 3(b), get executed once each. Since each line has an s/c of 1, the total step count for this case is 2. When $n > 1$, lines 3(a), 5, and 12 are each executed once. Line 6 gets executed n times, and lines 7–11 get executed $n-1$ times each (note that the last time line 6 is executed, i is incremented to $n+1$ and the loop exited). Line 5 has an s/c of 2; the remaining lines that get executed have an s/c of 1. The total steps for the case $n > 1$ is therefore $4n+1$. \square

```

1  void Fibonacci( int n)
2  { // compute the Fibonacci number  $F_n$ 
3    if (n <= 1) cout << n << endl; if  $F_0 = 0$  and  $F_1 = 1$ 
4    else { // compute  $F_n$ 
5        int f0; int fsum2 = 0; int fsum1 = 1;
6        for (int i = 2; i <= n; i++)
7        {
8            f0 = fsum1 + fsum2;
9            fsum2 = fsum1;
10           fsum1 = f0;
11        } // end of for
12        cout << f0 << endl;
13    } // end of else
14 }

```

Program 1.25: Fibonacci numbers

Summary

The time complexity of a program is given by the number of steps taken by the program to compute the function it was written for. The number of steps is itself a function of the instance characteristics. Although any specific instance may have several characteristics (e.g., the number of inputs, the number of outputs, the magnitudes of the inputs and outputs), the number of steps is computed as a function of some subset of these. Usually, we choose those characteristics that are of importance to us. For example, we might wish to know how the computing (or run) time (i.e., time complexity) increases as the number of inputs increase. In this case the number of steps will be computed as a function of the number of inputs alone. For a different program, we might be interested in determining how the computing time increases as the magnitude of one of the inputs increases. In this case the number of steps will be computed as a function of the magnitude of this input alone. Thus, before the step count of a program can be determined, we need to know exactly which characteristics of the problem instance are to be used. These define the variables in the expression for the step count. In the case of *Sum*, we chose to measure the time complexity as a function of the number, n , of elements being added. For function *Add*, the choice of characteristics was the number, m , of rows and the number, n , of columns in the matrices being added.

Once the relevant characteristics (n, m, p, q, r, \dots) have been selected, we can define what a step is. A step is any computation unit that is independent of the characteristics (n, m, p, q, r, \dots). Thus, 10 additions can be one step;

100 multiplications can also be one step; but n additions cannot. Nor can $m/2$ additions, $p+q$ subtractions, and so on be counted as one step.

A systematic way to assign step counts was also discussed. Once this has been done, the time complexity (i.e., the total step count) of a program can be obtained using either of the two methods discussed.

The examples we have looked at so far were sufficiently simple that the time complexities were nice functions of fairly simple characteristics like the number of elements, and the number of rows and columns. For many programs, the time complexity is not dependent solely on the number of inputs or outputs or some other easily specified characteristic. Consider the function *BinarySearch* (Program 1.10). This function searches $a[0], \dots, a[n-1]$ for x . A natural parameter with respect to which you might wish to determine the step count is the number, n , of elements to be searched. That is, we would like to know how the computing time changes as we change the number of elements n . The parameter n is inadequate. For the same n , the step count varies with the position of x in a . We can extricate ourselves from the difficulties resulting from situations wherein the chosen parameters are not adequate to determine the step count uniquely by defining three kinds of step counts: best-case, worst-case, and average.

The *best-case step count* is the minimum number of steps that can be executed for the given parameters. The *worst-case step count* is the maximum number of steps that can be executed for the given parameters. The *average step count* is the average number of steps executed on instances with the given parameters.

1.7.1.3 Asymptotic Notation (O , Ω , Θ)

Our motivation to determine step counts is to be able to compare the time complexities of two programs that compute the same function and also to predict the growth in run time as the instance characteristics change.

Determining the exact step count (either worst-case or average) of a program can prove to be an exceedingly difficult task. Expending immense effort to determine the step count exactly is not a very worthwhile endeavor, since the notion of a step is itself inexact. (Both the instructions $x = y$ and $x = y + z + (x/z) + (x*y*z - x/z)$ count as one step.) Because of the inexactness of what a step stands for, the exact step count is not very useful for comparative purposes. An exception to this is when the difference in the step counts of two programs is very large, as in $3n + 3$ versus $100n + 10$. We might feel quite safe in predicting that the program with step count $3n + 3$ will run in less time than the one with step count $100n + 10$. But even in this case, it is not necessary to know that the exact step count is $100n + 10$. Something like, "it's about $80n$, or $85n$, or $75n$," is adequate to arrive at the same conclusion.

52 Basic Concepts

For most situations, it is adequate to be able to make a statement like $c_1n^3 \leq t_p(n) \leq c_2n^3$ or $t_p(n, m) = c_1n + c_2m$ where c_1 and c_2 are nonnegative constants. This is so because if we have two programs with a complexity of $c_1n^3 + c_2n$ and c_3n respectively, then we know that the one with complexity c_3n will be faster than the one with complexity $c_1n^3 + c_2n$ for sufficiently large values of n . For small values of n , either program could be faster (depending on c_1 , c_2 , and c_3). If $c_1 = 1$, $c_2 = 2$, and $c_3 = 100$ then $c_1n^3 + c_2n \leq c_3n$ for $n \leq 98$, and $c_1n^3 + c_2n > c_3n$ for $n > 98$. If $c_1 = 1$, $c_2 = 2$, and $c_3 = 1000$, then $c_1n^3 + c_2n \leq c_3n$ for $n \leq 998$.

No matter what the values of c_1 , c_2 , and c_3 , there will be an n beyond which the program with complexity c_3n will be faster than the one with complexity $c_1n^3 + c_2n$. This value of n will be called the *break-even point*. If the break-even point is zero, then the program with complexity c_3n is always faster (or at least as fast). The exact break-even point cannot be determined analytically. The programs have to be run on a computer in order to determine the break-even point. To know that there is a break-even point, it is adequate to know that one program has complexity $c_1n^3 + c_2n$ and the other c_3n for some constants c_1 , c_2 , and c_3 . There is little advantage in determining the exact values of c_1 , c_2 , and c_3 .

With the previous discussion as motivation, we introduce some terminology that will enable us to make meaningful (but inexact) statements about the time and space complexities of a program. In the remainder of this chapter, the functions f and g are nonnegative functions.

Definition [Big "oh"]: $f(n) = O(g(n))$ (read as " f of n is big oh of g of n ") iff (if and only if) there exist positive constants c and n_0 such that $f(n) \leq cg(n)$ for all n , $n \geq n_0$. □

Example 1.15: $3n + 2 = O(n)$ as $3n + 2 \leq 4n$ for all $n \geq 2$. $3n + 3 = O(n)$ as $3n + 3 \leq 4n$ for all $n \geq 3$. $100n + 6 = O(n)$ as $100n + 6 \leq 101n$ for $n \geq 10$. $10n^2 + 4n + 2 = O(n^2)$ as $10n^2 + 4n + 2 \leq 11n^2$ for $n \geq 5$. $1000n^2 + 100n - 6 = O(n^2)$ as $1000n^2 + 100n - 6 \leq 1001n^2$ for $n \geq 100$. $6 \cdot 2^n + n^2 = O(2^n)$ as $6 \cdot 2^n + n^2 \leq 7 \cdot 2^n$ for $n \geq 4$. $3n + 3 = O(n^2)$ as $3n + 3 \leq 3n^2$ for $n \geq 2$. $10n^2 + 4n + 2 = O(n^4)$ as $10n^2 + 4n + 2 \leq 10n^4$ for $n \geq 2$. $3n + 2 \neq O(1)$ as $3n + 2$ is not less than or equal to c for any constant c and all n , $n \geq n_0$. $10n^2 + 4n + 2 \neq O(n)$. □

We write $O(1)$ to mean a computing time that is a constant. $O(n)$ is called linear, $O(n^2)$ is called quadratic, $O(n^3)$ is called cubic, and $O(2^n)$ is called exponential. If an algorithm takes time $O(\log n)$, it is faster, for sufficiently large n , than if it had taken $O(n)$. Similarly, $O(n \log n)$ is better than $O(n^2)$ but not as good as $O(n)$. These seven computing times, $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$, and $O(2^n)$ are the ones we will see most often in this book.

As illustrated by the previous example, the statement $f(n) = O(g(n))$ states only that $g(n)$ is an upper bound on the value of $f(n)$ for all $n, n \geq n_0$. It does not say anything about how good this bound is. Notice that $n = O(n^2)$, $n = O(n^{1.5})$, $n = O(n^3)$, $n = O(2^n)$, and so on. For the statement $f(n) = O(g(n))$ to be informative, $g(n)$ should be as small a function of n as one can come up with for which $f(n) = O(g(n))$. So, while we shall often say $3n + 3 = O(n)$, we shall almost never say $3n + 3 = O(n^2)$, even though this latter statement is correct.

From the definition of O , it should be clear that $f(n) = O(g(n))$ is not the same as $O(g(n)) = f(n)$. In fact, it is meaningless to say that $O(g(n)) = f(n)$. The use of the symbol " $=$ " is unfortunate because this symbol commonly denotes the "equals" relation. Some of the confusion that results from the use of this symbol (which is standard terminology) can be avoided by reading the symbol " $=$ " as "is" and not as "equals."

Theorem 1.2 obtains a very useful result concerning the order of $f(n)$ (i.e., the $g(n)$ in $f(n) = O(g(n))$) when $f(n)$ is a polynomial in n .

Theorem 1.2: If $f(n) = a_m n^m + \cdots + a_1 n + a_0$, then $f(n) = O(n^m)$.

$$\begin{aligned} \text{Proof: } f(n) &\leq \sum_{i=0}^m |a_i| n^i \\ &\leq n^m \sum_{i=0}^m |a_i| n^{i-m} \\ &\leq n^m \sum_{i=0}^m |a_i|, \text{ for } n \geq 1 \end{aligned}$$

So, $f(n) = O(n^m)$. \square

Definition: [Omega] $f(n) = \Omega(g(n))$ (read as " f of n is omega of g of n ") iff there exist positive constants c and n_0 such that $f(n) \geq cg(n)$ for all $n, n \geq n_0$. \square

Example 1.16: $3n + 2 = \Omega(n)$ as $3n + 2 \geq 3n$ for $n \geq 1$ (actually the inequality holds for $n \geq 0$, but the definition of Ω requires an $n_0 > 0$); $3n + 3 = \Omega(n)$ as $3n + 3 \geq 3n$ for $n \geq 1$; $100n + 6 = \Omega(n)$ as $100n + 6 \geq 100n$ for $n \geq 1$; $10n^2 + 4n + 2 = \Omega(n^2)$ as $10n^2 + 4n + 2 \geq n^2$ for $n \geq 1$; $6 \cdot 2^n + n^2 = \Omega(2^n)$ as $6 \cdot 2^n + n^2 \geq 2^n$ for $n \geq 1$. Observe also that $3n + 3 = \Omega(1)$; $10n^2 + 4n + 2 = \Omega(n)$; $10n^2 + 4n + 2 = \Omega(1)$; $6 \cdot 2^n + n^2 = \Omega(n^{100})$; $6 \cdot 2^n + n^2 = \Omega(n^{30.2})$; $6 \cdot 2^n + n^2 = \Omega(n^3)$; $6 \cdot 2^n + n^2 = \Omega(n)$; and $6 \cdot 2^n + n^2 = \Omega(1)$. \square

As in the case of the "big oh" notation, there are several functions $g(n)$ for which $f(n) = \Omega(g(n))$. The function $g(n)$ is only a lower bound on $f(n)$. For the

54 Basic Concepts

statement $f(n) = \Omega(g(n))$ to be informative, $g(n)$ should be as large a function of n as possible for which the statement $f(n) = \Omega(g(n))$ is true. So, while we shall say that $3n + 3 = \Omega(n)$ and $6 \cdot 2^n + n^2 = \Omega(2^n)$, we shall almost never say that $3n + 3 = \Omega(1)$ or $6 \cdot 2^n + n^2 = \Omega(1)$, even though both of these statements are correct.

Theorem 1.3 is the analogue of Theorem 1.2 for the omega notation.

Theorem 1.3: If $f(n) = a_m n^m + \cdots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Omega(n^m)$.

Proof: Left as an exercise. \square

Definition: [Theta] $f(n) = \Theta(g(n))$ (read as “ f of n is theta of g of n ”) iff there exist positive constants c_1 , c_2 , and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n , $n \geq n_0$. \square

Example 1.17: $3n + 2 = \Theta(n)$ as $3n + 2 \geq 3n$ for all $n \geq 2$, and $3n + 2 \leq 4n$ for all $n \geq 2$, so $c_1 = 3$, $c_2 = 4$, and $n_0 = 2$. $3n + 3 = \Theta(n)$; $10n^2 + 4n + 2 = \Theta(n^2)$; $6 \cdot 2^n + n^2 = \Theta(2^n)$; and $10 \cdot \log n + 4 = \Theta(\log n)$. $3n + 2 \neq \Theta(1)$; $3n + 3 \neq \Theta(n^2)$; $10n^2 + 4n + 2 \neq \Theta(n)$; $10n^2 + 4n + 2 \neq \Theta(1)$; $6 \cdot 2^n + n^2 \neq \Theta(n^2)$; $6 \cdot 2^n + n^2 \neq \Theta(n^{100})$; and $6 \cdot 2^n + n^2 \neq \Theta(1)$. \square

The theta notation is more precise than both the “big oh” and omega notations. $f(n) = \Theta(g(n))$ iff $g(n)$ is both an upper and lower bound on $f(n)$.

Notice that the coefficients in all of the $g(n)$'s used in the preceding three examples have been 1. This is in accordance with practice. We shall almost never find ourselves saying that $3n + 3 = \Theta(3n)$, or that $10 = \Theta(100)$, or that $10n^2 + 4n + 2 = \Theta(4n^2)$, or that $6 \cdot 2^n + n^2 = \Theta(6 \cdot 2^n)$, or that $6 \cdot 2^n + n^2 = \Theta(4 \cdot 2^n)$, even though each of these statements is true.

Theorem 1.4: If $f(n) = a_m n^m + \cdots + a_1 n + a_0$ and $a_m > 0$, then $f(n) = \Theta(n^m)$.

Proof: Left as an exercise. \square

Let us reexamine the time complexity analyses of the previous section. For function *Sum* (Program 1.17) we had determined that $t_{\text{sum}}(n) = 2n + 3$. So, $t_{\text{sum}}(n) = \Theta(n)$, $t_{\text{sum}}(n) = 2n + 2 = \Theta(n)$ and $t_{\text{sum}}(n, n) = 2nn + 2n + 1 = \Theta(nn)$.

Although we might all see that the Ω , Ω , and Θ notations have been used correctly in the preceding paragraphs, we are still left with the question, Of what use are these notations if one has to first determine the step count exactly? The answer to this question is that the asymptotic complexity (i.e., the complexity in terms of Ω , Ω , and Θ) can be determined quite easily without determining the exact step count. This is usually done by first determining the asymptotic

complexity of each statement (or group of statements) in the program and then adding up these complexities. Tables 1.4 through 1.6 do just this for *Sum*, *Rsum*, and *Add* (Programs 1.17, 1.18, and 1.22).

Note that in the table for *Add* (Table 1.6), lines 3 and 4 have been lumped together even though they have different frequencies. This lumping together of these two lines is possible because their frequencies are of the same order.

line(s)	s/e	frequency	total steps
1	0	—	$\Theta(0)$
2	1	1	$\Theta(1)$
3	1	$n+1$	$\Theta(n)$
4	1	n	$\Theta(n)$
5	1	1	$\Theta(1)$
6	0	—	$\Theta(0)$
$t_{\text{Sum}}(n) = \Theta(\max_{1 \leq i \leq 6} \{g_i(n)\}) = \Theta(n)$			

Table 1.4: Asymptotic complexity of *Sum* (Program 1.17)

line	s/e	frequency		total steps	
		$n = 0$	$n > 0$	$n = 0$	$n > 0$
1	0	—	—	0	$\Theta(0)$
2(a)	1	1	1	1	$\Theta(1)$
2(b)	1	1	0	1	$\Theta(0)$
3	$2 + t_{\text{Rsum}}(n-1)$	0	1	0	$\Theta(2 + t_{\text{Rsum}}(n-1))$
4	0	—	—	0	$\Theta(0)$
$t_{\text{Rsum}}(n) =$				2	$\Theta(2 + t_{\text{Rsum}}(n-1))$

Table 1.5: Asymptotic complexity of *Rsum* (Program 1.18)

Although the analyses of Tables 1.4 through 1.6 are actually carried out in terms of step counts, it is correct to interpret $t_P(n) = \Theta(g(n))$, or $t_P(n) = O(g(n))$, or $t_P(n) = \Omega(g(n))$ as a statement about the computing time of program *P*. This is so because each step takes only $\Theta(1)$ time to execute.

After you have had some experience using the table method, you will be in a position to arrive at the asymptotic complexity of a program by taking a more

$\text{line}(s)$	s/t	frequency	total steps
1	0	—	$\Theta(1)$
2	1	$\Theta(n)$	$\Theta(n)$
3,4	1	$\Theta(nn)$	$\Theta(nn)$
5	0	—	$\Theta(1)$
$t_{\text{Add}}(n, n) =$			$\Theta(nn)$

Table 1.6: Asymptotic complexity of *Add* (Program 1.22)

global approach. We elaborate on this method in the following examples.

Example 1.18 [Permutation generator]: Consider function *Permutations* (Program 1.12). Assume that a is of size n . When $k = n-1$, we see that the time taken is $\Theta(n)$. When $k < n-1$, the else clause is entered. At this time, the second for loop is entered $n-k$ times. Each iteration of this loop takes $\Theta(t_{\text{Permutations}}(k+1, n-1))$ time. So, $t_{\text{Permutations}}(k, n-1) = \Theta((n-k)t_{\text{Permutations}}(k+1, n-1))$ when $k < n-1$. Using the substitution method, we obtain $t_{\text{Permutations}}(0, n-1) = \Theta(n(n!))$, $n \geq 1$. \square

Example 1.19 [Binary search]: Let us obtain the time complexity of function *BinarySearch* (Program 1.10). The instance characteristic that we shall use is the number n of elements in a . Each iteration of the for loop takes $\Theta(1)$ time. We can show that the for loop is iterated at most $\lceil \log_2(n+1) \rceil$ times. Since an asymptotic analysis is being performed, we do not need such an accurate count of the worst-case number of iterations. Each iteration except for the last results in a decrease in the size of the segment of a that has to be searched by a factor of about 2. So, this loop is iterated $\Theta(\log n)$ times in the worst-case. As each iteration takes $\Theta(1)$ time, the overall worst-case complexity of *BinarySearch* is $\Theta(\log n)$. Note that, if a was passed by value, the complexity of using *BinarySearch* would be more than this because it would take $\Omega(n)$ time just to invoke the function. \square

Example 1.20 [Magic square]: Our final example is a problem from recreational mathematics. A magic square is an $n \times n$ matrix of the integers 1 to n^2 such that the sum of every row, column, and diagonal is the same. Figure 1.3 gives an example magic square for the case $n = 5$. In this example, the common sum is 65.

H. Coseter has given the following simple rule for generating a magic

15	8	1	24	17
16	14	7	5	23
22	20	13	6	4
3	21	19	12	10
9	2	25	18	11

Figure 1.3: Example magic square

square when n is odd:

Start with 1 in the middle of the top row; then go up and left, assigning numbers in increasing order to empty squares; if you fall off the square imagine the same square as tiling the plane and continue; if a square is occupied, move down instead and continue.

The magic square of Figure 1.3 was formed using this rule. Program 1.26 is the C++ program for creating an $n \times n$ magic square for the case when n is odd. This results from Coxeater's rule.

The magic square is represented using a two-dimensional array having n rows and n columns. For this application it is convenient to number the rows (and columns) from zero to $n - 1$ rather than from one to n . Thus, when the program "falls off the square," i and/or j are set back to zero or $n - 1$.

The while loop is governed by the variable *key*, which is an integer variable initialized to 2 and increased by one each time through the loop. Thus, each statement within the while loop will be executed no more than $n^2 - 1$ times, and the computing time for *Magic* is $O(n^2)$. Since there are n^2 positions in which the algorithm must place a number, we see that $O(n^2)$ is the best bound on algorithms for the magic square problem can have. \square

58 Basic Concepts

```

void Magic( const int n)
// Create a magic square of size n, n is odd.
    const int MaxSize = 51; // maximum square size
    int square[MaxSize][MaxSize], k, i;

    // check correctness of n
    if (n > MaxSize) || (n < 1))
        throw "Error!..n out of range";
    else if ((n%2)) throw "Error!..n is even";

    // n is odd. Coester's rule can be used
    for (int i = 0; i < n; i++) // initialize square to 0
        fill(square[i], square[i] + n, 0); // STL algorithm
    square[0][(n-1)/2] = 1; // middle of first row

    // i and j are current position
    int key = 2; i = 0; int j = (n-1)/2;
    while (key <= n * n) {
        // move up and left
        if (i - 1 < 0) k = n - 1; else k = i - 1;
        if (j - 1 < 0) l = n - 1; else l = j - 1;
        if (square[k][l] != 0) i = (i+1)%n; // square occupied, move down
        else { // square [k][l] is unoccupied
            i = k; j = l;
        }
        square[i][j] = key;
        key++;
    } // end of while

    // output the magic square
    cout << "magic square of size " << n << endl;
    for (j = 0; j < n; j++) {
        copy(square[j], square[j] + n, ostream_iterator<int>(cout, " "));
        cout << endl;
    }
}

```

Program 1.26: Magic square

1.7.1.4 Practical Complexities

We have seen that the time complexity of a program is generally some function of the instance characteristics. This function is very useful in determining how the time requirements vary as the instance characteristics change. The complexity function may also be used to compare two programs P and Q that perform the same task. Assume that program P has complexity $\Theta(n)$ and program Q is of complexity $\Theta(n^2)$. We can assert that program P is faster than program Q for sufficiently large n . To see the validity of this assertion, observe that the actual computing time of P is bounded from above by cn for some constant c and for all $n, n \geq n_1$, whereas that of Q is bounded from below by dn^2 for some constant d and all $n, n \geq n_2$. Since $cn \leq dn^2$ for $n \geq c/d$, program P is faster than program Q whenever $n \geq \max\{n_1, n_2, c/d\}$.

You should always be cautiously aware of the presence of the phrase "sufficiently large" in the assertion of the preceding discussion. When deciding which of the two programs to use, we must know whether the n we are dealing with is, in fact, sufficiently large. If program P actually runs in $10^6 n$ milliseconds, whereas program Q runs in n^2 milliseconds, and if we always have $n \leq 10^6$, then, other factors being equal, program Q is the one to use.

To get a feel for how the various functions grow with n , you are advised to study Table 1.7 and Figure 1.4 very closely. It is evident from the table and the figure that the function 2^n grows very rapidly with n . In fact, if a program needs 2^n steps for execution, then when $n = 40$, the number of steps needed is approximately $1.1 \cdot 10^{12}$. On a computer performing 1 billion steps per second, this would require about 18.3 minutes. If $n = 50$, the same program would run for about 13 days on this computer. When $n = 60$, about 310.36 years will be required to execute the program and when $n = 100$, about $4 \cdot 10^{33}$ years will be needed. So, we may conclude that the utility of programs with exponential complexity is limited to small n (typically $n \leq 40$).

$\log n$	n	$n \log n$	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65,536
5	32	160	1024	32,768	4,294,967,296

Table 1.7: Function values

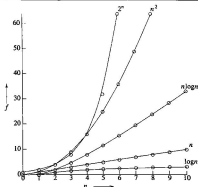


Figure 1.4: Plot of function values

Programs that have a complexity that is a polynomial of high degree are also of limited utility. For example, if a program needs n^{10} steps, then using our 1-billion-steps-per-second computer, we will need 10 seconds when $n = 10$; 3,171 years when $n = 100$; and $3.17 \cdot 10^{13}$ years when $n = 1000$. If the program's complexity had been n^5 steps instead, then we would need 1 second when $n = 1000$; 110.67 minutes when $n = 10,000$; and 11.57 days when $n = 100,000$.

Table 1.8 gives the time needed by a 1-billion-steps-per-second computer to execute a program of complexity $f(n)$ instructions. From a practical standpoint, it is evident that for reasonably large n (say $n > 100$), only programs of small complexity (such as n , $n \log n$, n^2 , n^3) are feasible. Further, this is the case even if one could build a computer capable of executing 10^{12} instructions per second. In this case, the computing times of Table 1.8 would decrease by a

n	$f(n)$						
	n	$n \log_2 n$	n^2	n^3	n^4	n^{10}	2^n
10	.01 μ s	.02 μ s	.1 μ s	1 μ s	10 μ s	10 s	1 μ s
20	.02 μ s	.09 μ s	.4 μ s	8 μ s	160 μ s	2.04 h	1 ms
30	.03 μ s	.15 μ s	.9 μ s	27 μ s	810 μ s	5.83 d	1 s
40	.04 μ s	.21 μ s	1.6 μ s	64 μ s	2.56 ms	121 d	16 μ s
50	.05 μ s	.28 μ s	2.5 μ s	125 μ s	6.25 ms	3.1 y	13 d
100	.10 μ s	.66 μ s	10 μ s	1 ms	100 ms	2071 y	4×10^{31} y
10^2	1 μ s	9.96 μ s	1 ms	1 s	16.67 m	3.17×10^{32} y	32×10^{32} y
10^3	10 μ s	130 μ s	100 ms	16.67 s	115.2 d	3.17×10^{33} y	
10^4	100 μ s	1.66 ms	10 s	11.57 d	3171 y	3.17×10^{34} y	
10^5	1 ms	16.92 ms	16.67 m	31.71 y	3.17×10^{35} y	3.17×10^{35} y	

μ = microsecond = 10^{-6} seconds; ms = milliseconds = 10^{-3} seconds
 s = seconds; m = minutes; h = hours; d = days; y = years

Table 1.8: Times on a 1-billion-steps-per-second computer

factor of 1000. Now, when $n = 100$ it would take 3.17 years to execute n^{10} instructions and 4×10^{31} years to execute 2^n instructions.

1.7.2 Performance Measurement

Performance measurement is concerned with obtaining the actual space and time requirements of a program. These quantities are dependent on the particular compiler and options used as well as on the specific computer on which the program is run. So, when you repeat the performance experiments reported in this book, the run times you will observe will be quite different. In fact, as computers are continuously getting faster, your times will, most likely, be much smaller than those reported in this book.

In keeping with the discussion of the preceding section, we shall not concern ourselves with the space and time needed for compilation. We justify this by the assumption that each program (after it has been fully debugged) will be compiled once and then executed several times. Certainly, the space and time needed for compilation are important during program testing, when more time is spent on this task than in actually running the compiled code.

We shall not explicitly consider measuring the run-time space requirements of a program. Rather, we shall focus on measuring the computing time of a program. To obtain the computing (or run) time of a program, we need a clocking

6.2. Basic Concepts

function. We assume the existence of a function `time(here)` that returns in the variable `time` the current time in hundredths of a second.

Suppose we wish to measure the worst-case performance of the sequential search function (Program 1.27). Before we can do this, we need to: (1) decide on the values of n for which the times are to be obtained and (2) determine, for each of the above values of n , the data that exhibits the worst-case behavior.

```
int SequentialSearch (int *a, const int n, const int x)
{ // Search a[0:n-1].
  int i;
  for (i = 0; i < n && a[i] != x; i++);
  if (i == n) return -1;
  else return i;
}
```

Program 1.27: Sequential search

The decision on which values of n to use is to be based on the amount of timing we wish to perform and also on what we expect to do with the times once they are obtained. Assume that for Program 1.27, our intent is simply to predict how long it will take, in the worst-case, to search for x given the size n of a . An asymptotic analysis reveals that this time is $\Theta(n)$. So, we expect a plot of the times to be a straight line. Theoretically, if we know the times for any two values of n , the straight line is determined, and we can obtain the time for all other values of n from this line. In practice, we need the times for more than two values of n . This is so for the following reasons:

- (1) Asymptotic analysis tells us the behavior only for “sufficiently large” values of n . For smaller values of n the run time may not follow the asymptotic curve. To determine the point beyond which the asymptotic curve is followed, we need to examine the times for several values of n .
- (2) Even in the region where the asymptotic behavior is exhibited, the times may not lie exactly on the predicted curve (straight line in the case of Program 1.27) because of the effects of low-order terms that are discarded in the asymptotic analysis. For instance, a program with asymptotic complexity $\Theta(n)$ can have an actual complexity that is $c_1n + c_2\log n + c_3$, or for that matter any other function of n in which the highest-order term is c_1n for some constant, $c_1, c_1 > 0$.

It is reasonable to expect that the asymptotic behavior of Program 1.27 will begin for some n that is smaller than 100. So, for $n > 100$ we shall obtain the run

time for just a few values. A reasonable choice is $n = 200, 300, 400, \dots, 1000$. There is nothing magical about this choice of values. We can just as well use $n = 500, 1000, 1500, \dots, 10,000$ or $n = 512, 1024, 2048, \dots, 2^{12}$. It will cost us more in terms of computer time to use the latter choices, and we will probably not get any better information about the run time of Program 1.27 using these choices.

For n in the range $[0, 100]$ we shall carry out a more refined measurement, since we are not quite sure where the asymptotic behavior begins. Of course, if our measurements show that the straight-line behavior does not begin in this range, we shall have to perform a more detailed measurement in the range $[100, 200]$ and so on, until the onset of this behavior is detected. Times in the range $[0, 100]$ will be obtained in steps of 10 beginning at $n = 0$.

It is easy to see that Program 1.27 exhibits its worst-case behavior when x is chosen such that it is not one of the $a[i]$'s. For definiteness, we shall set $a[i] = i$, $1 \leq i \leq n$ and $x = 0$.

At this time, we envision using a program such as Program 1.28 to obtain the worst-case times.

```

void TimeSearch() {
    int a[1001], n[20];
    for (int j = 1; j <= 1000; j++) // initialize a
        a[j] = j;
    for (j = 0; j < 10; j++) // values of n
        n[j] = 10 * j; n[j+10] = 100 * (j + 1);
}
cout << "  n  time" << endl;
for (j = 0; j < 20; j++) { // obtain computing times
    long start, stop;
    time(start); // start timer
    int k = SequentialSearch(a, n[j], 0); // unsuccessful search
    time(stop); // stop timer
    long runTime = stop - start;
    cout << "    " << n[j] << "    " << runTime << endl;
}
cout << "Times are in hundredths of a second." << endl;
}

```

Program 1.28: Program to time Program 1.27

The output obtained from this program is summarized in Figure 1.5. All the times are zero, indicating that the precision of our clock is inadequate.

n	time	n	time
0	0	100	0
10	0	200	0
20	0	300	0
30	0	400	0
40	0	500	0
50	0	600	0
60	0	700	0
70	0	800	0
80	0	900	0
90	0	1000	0

Times in hundredths of a second

Figure 1.5: Output from Program 1.28

To time a short event, it is necessary to repeat it several times and divide the total time for the event by the number of repetitions.

Since our clock has an accuracy of about one-hundredth of a second, we should not attempt to time any single event that takes less than about 1 second. With an event time of at least 1 second, we can expect our observed times to be accurate to 1 percent.

The body of Program 1.28 needs to be changed to that of Program 1.29. In this program, $r[j]$ is the number of times the search is to be repeated when the number of elements in the array is $n[j]$. Notice that rearranging the timing statements as in Programs Program 1.30 or Program 1.31 does not produce the desired results. For instance, from the data of Figure 1.5, we expect that with the structure of Program 1.30, the value output will still be 0 because in each iteration of the for loop, 0 gets added to *total*. With the structure of Program 1.31, we expect the program never to exit the for loop. Yet another alternative is to move the first call to *time* out of the for loop of Program 1.31 and change the assignment to *total* within the for loop to

$$\text{long total} = \text{stop} - \text{start};$$

This approach can be expected to yield satisfactory times. This approach cannot be used when the timing function available gives us only the time since the last invocation of *time*. Another difficulty is that the measured time includes the time needed to read the clock. For small n , this time may be larger than the time to run *SequentialSearch*. This difficulty can be overcome by determining the time

taken by the timing function and subtracting this time later. In further discussion, we shall use the explicit repetition factor technique.

```

void TimeSearch() {
    int a[1001], n[20];
    const long r[20] = { 300000, 300000, 200000, 200000, 100000, 100000,
        100000, 80000, 80000, 50000, 50000, 25000, 15000, 15000, 10000, 7500, 7000,
        6000, 5000, 5000 };

    for (int j = 1; j <= 1000; j++) // initialize a
        a[j] = j;

    for (j = 0; j < 10; j++) // values of n
        n[j] = 10 * j; n[j+10] = 100 * (j+1);
}

cout << "  n  totalTime  runTime" << endl;

for(j = 0; j < 20; j++) { // obtain computing times
    long start, stop;
    time(start); // start timer
    for (long b = 1; b <= r[j]; b++)
        int k = SequentialSearch(a, n[j], 0); // unsuccessful search
    time(stop); // stop timer
    long totalTime = stop - start;
    float runTime = (float) (totalTime)/(float)(r[j]);
    cout << "  " << n[j] << "  " << totalTime << "  " << runTime << endl;
}
cout << "Times are in hundredths of a second." << endl;
}

```

Program 1.29: Timing program

The output from the timing program, Program 1.29, is given in Figure 1.6. The times for n in the range $[0, 100]$ are plotted in Figure 1.7. The remaining values have not been plotted because this would lead to severe compression of the range $[0, 100]$. The linear dependence of the worst-case time on n is apparent from this graph.

The graph of Figure 1.7 can be used to predict the run time for other values of n . For example, we expect that when $n = 24$, the worst-case search time will be 0.0031 hundredths of a second. We can go one step further and get the

66 Basic Concepts

```
long total = 0;
for (long b = 1; b <= r[j]; b++)
{
    time (start);
    k = SequentialSearch(a, n[j], 0);
    time (stop);
    total += stop - start;
}
float runTime = (float) (total) / (float) (r[j]);
```

Program 1.30: Improper timing construct

```
for(long total = 0, i = 0; total < DerivedTime; i++)
{
    time (start);
    k = SequentialSearch(a, n[j], 0);
    time (stop);
    total += stop - start;
}
float runTime = (float) (total) / (float) (i);
```

Program 1.31: Another improper timing construct

equation of the straight line. The equation of this line is $t = c + mn$, where m is the slope and c the value for $n = 0$. From the graph, we see that $c = 0.0008$. Using the point $n = 60$ and $t = 0.0066$, we obtain $m = (t - c)/n = 0.0058/60 = 0.000096$. So, the line of Figure 1.7 has the equation $t = 0.0008 + 0.000096n$, where t is the time in hundredths of a second. From this, we expect that when $n = 1000$, the worst-case search time will be 0.0978 msec, and when $n = 500$, it will be 0.0491 msec. Compared with the actual observed times of Figure 1.6, we see that these figures are very accurate!

An alternate approach to obtain a good straight line for the data of Figure 1.6 is to obtain the straight line that is the least-squares approximation to the data. The result is $t = 0.00085434 + 0.00009564n$. When $n = 1000$ and 500, this equation yields $t = 0.0966$ and 0.0487.

Now, we are probably ready to put ourselves on the back for a job well done. However, this action is somewhat premature, since our experiment is

<i>n</i>	<i>total</i>	<i>runTime</i>	<i>n</i>	<i>total</i>	<i>runTime</i>
0	241	0.0008	100	527	0.0105
10	333	0.0018	200	505	0.0202
20	382	0.0029	300	451	0.0301
30	736	0.0037	400	593	0.0395
40	467	0.0047	500	494	0.0494
50	565	0.0056	600	439	0.0585
60	639	0.0066	700	484	0.0691
70	604	0.0075	800	467	0.0778
80	681	0.0085	900	434	0.0868
90	472	0.0094	1000	484	0.0968

Times in hundredths of a second

Figure 1.6: Worst-case run times for Program 1.27

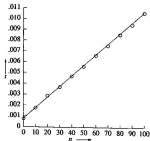


Figure 1.7: Plot of the data in Figure 1.6

68 Basic Concepts

flawed. First, the measured time includes the time taken by the repetition for loop. So, the times of Figure 1.6 are excessive. This can be corrected by determining the time for each iteration of this statement. A quick test run indicates that 300,000 executions take only 50 hundredths of a second. So, subtracting the time for the for ($b = 1$; $b \leq r[j]$; $b++$) statement reduces the reported times by only 0.000016. We can ignore this difference, since the use of a higher repetition factor could well result in measured times that are lower by about 0.000016 hsec per search.

Summary

To obtain the run time of a program, we need to plan the experiment. The following issues need to be addressed during the planning stage:

- (1) What is the accuracy of the clock? How accurate do our results have to be? Once the desired accuracy is known, we can determine the length of the shortest event that should be timed.
- (2) For each instance size, a repetition factor needs to be determined. This is to be chosen such that the event time is at least the minimum time that can be clocked with the desired accuracy.
- (3) Are we measuring worst-case or average performance? Suitable test data need to be generated.
- (4) What is the purpose of the experiment? Are the times being obtained for comparative purposes, or are they to be used to predict actual run times? If the latter is the case, then contributions to the run time from such sources as the repetition loop and data generation need to be subtracted (in case they are included in the measured time). If the former is the case, then these times need not be subtracted (provided they are the same for all programs being compared).
- (5) In case the times are to be used to predict actual run times, then we need to fit a curve through the points. For this, the asymptotic complexity should be known. If the asymptotic complexity is linear, then a least-squares straight line can be fit; if it is quadratic, then a parabola is to be used (i.e., $t = a_0 + a_1n + a_2n^2$). If the complexity is $\Theta(n \log n)$, then a least-squares curve of the form $t = a_0 + a_1n + a_2n \log_2 n$ can be fit. When obtaining the least-squares approximation, one should discard data corresponding to “small” values of n , since the program does not exhibit its asymptotic behavior for these n .

1.7.3 Generating Test Data

Generating a data set that results in the worst-case performance of a program is not always easy. In some cases, it is necessary to use a computer program to generate the worst-case data. In other cases, even this is very difficult. In these cases, another approach to estimating worst-case performance is taken. For each set of values of the instance characteristics of interest, we generate a suitably large number of random test data. The run times for these test data are obtained. The maximum of these times is used as an estimate of the worst-case time for this set of values of the instance characteristics.

To measure average-case times, it is usually not possible to average over all possible instances of a given characteristic. Although it is possible to do this for sequential and binary search, it is not possible for a sort program. If we assume that all keys are distinct, then for any given n , $n!$ different permutations need to be used to obtain the average time. Obtaining average-case data is usually much harder than obtaining worst-case data. So, we often adopt the strategy outlined above and simply obtain an estimate of the average time.

Whether we are estimating worst-case or average time using random data, the number of instances that we can try is generally much smaller than the total number of such instances. Hence, it is desirable to analyze the algorithm being tested to determine classes of data that should be generated for the experiment. This is a very algorithm-specific task, and we shall not discuss it here.

EXERCISES

1. Compare the two functions n^3 and $2^n/4$ for various values of n . Determine when the second becomes larger than the first.
2. Prove by induction:
 - (a) $\sum_{i=1}^n i = n(n+1)/2, n \geq 1$
 - (b) $\sum_{i=1}^n i^2 = n(n+1)(2n+1)/6, n \geq 1$
 - (c) $\sum_{i=0}^n x^i = (x^{n+1} - 1)/(x - 1), x \neq 1, n \geq 0$
3. Determine the frequency counts for all statements in the following two program segments:

70 Basic Concepts

```

1 for (i = 1; i <= n; i++)
2   for (j = 1; j <= i; j++)
3     for (k = 1; k <= j; k++)
4       x++;

```

(a)

```

1 i = 1;
2 while (i <= n)
3 {
4   x++;
5   i++;
6 }

```

(b)

4. (a) Introduce statements to increment *count* at all appropriate points in Program 1.32.

```

void D(int *x, int n)
{
    int i = 1;
    do {
        x[i] += 2;
        i += 2;
    }
    while (i <= n);
    i = 1;
    while (i <= (n/2))
    {
        x[i] += x[i+1];
        i++;
    }
}

```

Program 1.32: Example program

- (b) Simplify the resulting program by eliminating statements. The simplified program should compute the same value for *count* as computed by the program of (a).
 - (c) What is the exact value of *count* when the program terminates? You may assume that the initial value of *count* is 0.
 - (d) Obtain the step count for Program 1.32 using the frequency method. Clearly show the step count table.
5. Do Exercise 4 for function *Transpose* (Program 1.33).
 6. Do Exercise 4 for Program 1.34. This program multiplies two $n \times n$ matrices *a* and *b*.

```

void Transpose(int **a, int n)
{
    for (int i = 0; i < n-1; i++)
        for (int j = i+1; j < n; j++)
            swap(a[i][j], a[j][i]);
}

```

Program 1.33: Matrix transpose

```

void Multiply( int **a, int **b, int **c, int n)
{
    for (int i = 0; i < n ; i++)
        for (int j = 0; j < n ; j++)
        {
            c[i][j] = 0;
            for (int k = 0; k < n ; k++)
                c[i][j] += a[i][k] * b[k][j];
        }
}

```

Program 1.34: Square matrix multiplication

7. (a) Do Exercise 4 for Program 1.35. This program multiplies two matrices a and b where a is an $m \times n$ matrix and b is an $n \times p$ matrix.
- (b) Under what conditions will it be profitable to interchange the two outermost for loops?

8. Show that the following equalities are correct:

(a) $5n^2 - 6n = \Theta(n^2)$

(b) $n! = \Theta(n^n)$

(c) $2n^22^n + n \log n = \Theta(n^22^n)$

(d) $\sum_{i=0}^n i^2 = \Theta(n^3)$

(e) $\sum_{i=0}^n i^3 = \Theta(n^4)$

(f) $n^{2^x} + 6 \cdot 2^n = \Theta(2^{2^x})$

9
10

72 Basic Concepts

```

void Multiply(int **a, int **b, int **c, int m, int n, int p)
{
    for (int i = 0; i < m; i++)
        for (int j = 0; j < p; j++)
        {
            c[i][j] = 0;
            for (int k = 0; k < n; k++)
                c[i][j] += a[i][k] * b[k][j];
        }
}

```

Program 1.35: Matrix multiplication

- (g) $n^3 + 10^6 n^2 = \Theta(n^3)$
 - (h) $6n^3(\log n + 1) = O(n^3)$
 - (i) $n^{1.001} + n \log n = \Theta(n^{1.001})$
 - (j) $n^k + n^k \log n = \Theta(n^k + n)$ for all k and n , $k \geq 0$, and $n > 0$
 - (k) $10n^3 + 15n^4 + 100n^2 2^n = O(100n^2 2^n)$
 - (l) $33n^3 + 4n^3 = \Omega(n^3)$
 - (m) $33n^3 + 4n^2 = \Omega(n^3)$
9. Show that the following equalities are incorrect:
 - (a) $10n^2 + 9 = O(n)$
 - (b) $n^2 \log n = \Theta(n^3)$
 - (c) $n^3 \log n = \Theta(n^2)$
 - (d) $n^3 2^n + 6n^2 3^n = O(n^3 2^n)$
 10. Obtain the average run time of function *BinarySearch* (Program 1.10). Do this for suitable values of n in the range $[0, 100]$. Your report must include a plan for the experiment as well as the measured times. These times are to be provided both in a table and as a graph.
 11. Analyze the computing time of function *SelectionSort* (Program 1.8).
 12. Obtain worst-case run times for function *SelectionSort* (Program 1.8). Do this for suitable values of n in the range $[0, 3000]$. Your report must include a plan for the experiment as well as the measured times. These times are to be provided both in a table and as a graph.

13. Consider function *Add* (Program 1.22).
 - (a) Obtain run times for $n = 100, 200, \dots, 3000$.
 - (b) Plot the times obtained in part (a).
14. Do the previous exercise for matrix multiplication (Program 1.35).
15. A complex-valued matrix X is represented by a pair of matrices (A, B) where A and B contain real values. Write a program that computes the product of two complex-valued matrices (A, B) and (C, D) , where $(A, B) * (C, D) = (A + iB) * (C + iD) = (AC - BD) + i(AD + BC)$. Determine the number of additions and multiplications if the matrices are all $n \times n$.
16. Function *Magic* (Program 1.26) uses a 51×51 array *square* independent of the value of n . When $n < 51$, excess space is used and when $n > 51$, the function throws an exception. We can eliminate these shortcomings by using a dynamically allocated $n \times n$ 2-dimensional array. Modify function *Magic* to do this. Test your code.

1.8 REFERENCES AND SELECTED READINGS

A good introduction to programming in C++ can be found in the texts *C++ Program Design: An Introduction to Programming and Object-Oriented Design* by J. Cohoon and J. Davidson, 3rd Edition, McGraw Hill, NY, 2002 and *C++ How to Program*, 4th Edition, by H. Deitel and P. Deitel, Prentice Hall, Englewood Cliffs, NJ, 2002. For advanced C++ programming techniques, see *Effective C++*, Third Edition, by Scott Meyers, Addison-Wesley, Reading, MA, 2005. You can find a description of all components of the STL at the Web site <http://codeguru.com/earthweb.com/stl/stlguide> or *The C++ Standard Library: A Tutorial and Reference*, by N. Josuttis, Addison-Wesley, New York, 1999. For a discussion on the object-oriented paradigm, see *Object Oriented Analysis and Design with Applications*, Second Edition, by Grady Booch, Addison-Wesley, 1995.

The Art of Software Testing by G. Myers, John Wiley, New York, NY, 1979 and *Software Testing Techniques* by B. Beizer, Second Edition, Van Nostrand Reinhold, New York, NY, 1990 have more thorough treatments of software testing and debugging techniques.

The following books provide asymptotic analyses for several programs: *Fundamentals of Computer Algorithms* by E. Horowitz, S. Sahni, and S. Rajasekaran, W. H. Freeman and Co., New York, NY, 1998; *Introduction to Algorithms*, Second Edition, by T. Cormen, C. Leiserson, and R. Rivest, McGraw-Hill, New York, NY, 2002; and *Compared to What: An Introduction to the Analysis of Algorithms* by G. Rawlins, W. H. Freeman and Co., NY, 1992.

CHAPTER 2

Arrays

2.1 ABSTRACT DATA TYPES AND THE C++ CLASS

2.1.1 An Introduction to the C++ Class

C++ provides an explicit mechanism, the *class*, to support the distinction between specification and implementation and to hide the implementation of an ADT from its users. However, it is the programmer's responsibility to use the class mechanism judiciously so that it does, in fact, represent an ADT. The C++ class consists of four components (Program 2.1):

- (1) A class name: (e.g., *Rectangle*).
- (2) Data members: the data that makes up the class (e.g., *xLow*, *yLow*, *height*, and *width*).
- (3) Member functions: the set of operations that may be applied to the objects

of a class (e.g., *GetHeight()*, *GetWidth()*).

(4) Levels of program access: these control the level of access to data members and member functions from program code that is outside the class. There are three levels of access to class members: public, protected, and private.

Any public data member (member function) can be accessed (invoked) from anywhere in the program. A private data member (member function) can only be accessed (invoked) from within its class or by a function or a class that is declared to be a friend. A protected data member (member function) can only be accessed (invoked) from within its class or from its subclasses or by a friend. We will discuss subclasses when we study inheritance in Chapter 3.

```
#ifndef RECTANGLE_H
#define RECTANGLE_H
// In the header file Rectangle.h
class Rectangle {
public:    // the following members are public
    // the next four members are member functions
    Rectangle();    // constructor
    ~Rectangle();    // destructor
    int GetHeight();    // returns the height of the rectangle
    int GetWidth();    // returns the width of the rectangle
private: // the following members are private
    // the following members are data members
    int xLow, yLow, height, width;
    // (xLow, yLow) are the coordinates of the bottom left corner of the rectangle
};
#endif
```

Program 2.1: Definition of the C++ class *Rectangle*

2.1.2 Data Abstraction and Encapsulation in C++

Data encapsulation is enforced in C++ by declaring all data members of a class to be private (or protected). External access to data members, if required, can be achieved by defining member functions that get and set data members. In Program 2.1, *GetHeight()* and *GetWidth()* are used to access the private data members *height* and *width*. Member functions that will be invoked externally are declared public; all others are declared private or protected.

76 Arrays

Next, we discuss how the specification of the operations of a class is separated from their implementation in C++. The specification, which must be contained inside the public portion of the class definition of the ADT, consists of the names of every public member function, the type of its arguments, and the type of its result (This information about a function is known as its *function prototype*). There should also be a description of what the function does, which does not refer to the internal representation or implementation details. This requirement is quite important because it implies that an abstract data type is *implementation-independent*. This description may be achieved in C++ by using comments to describe what each member function does (like the DVD instruction manual mentioned in Chapter 1). Finally, the specification of an operation is physically separated from its implementation by placing it in an appropriately named header file (e.g., the contents of Program 2.1 are placed in *Rectangle.h*). The implementations of the functions are typically placed in a source file of the same name (e.g., the contents of Program 2.2 are placed in *Rectangle.cpp*). Note that C++ syntax does allow you to include the implementation of a member function inside its class definition. In this case, the function is treated as an *inline* function.

```
// In the source file Rectangle.cpp
#include "Rectangle.h"
```

```
// The prefix "Rectangle::" identifies GetHeight() and GetWidth() as member
// functions belonging to class Rectangle. It is required because the member
// functions are implemented outside their class definition
```

```
int Rectangle::GetHeight() { return height;}
int Rectangle::GetWidth() { return width;}
```

Program 2.2: Implementation of operations on *Rectangle*

2.1.3 Declaring Class Objects and Invoking Member Functions

Program 2.3 shows a fragment of code that illustrates how class objects are declared and how member functions that operate on these class objects are invoked. Class objects are declared and created in the same way that variables are declared or created. Members of an object are accessed or invoked by using the component selection operators, a dot (.) for direct component selection and an arrow (→), which we shall write as →, for indirect component selection

through a pointer.

```
// In a source file main.cpp
#include <iostream>
#include "Rectangle.h"

main() {
    Rectangle r, s;           // r and s are objects of class Rectangle
    Rectangle *i = &s;       // i is a pointer to class object s
    .
    .
    // use . to access members of class objects.
    // use -> to access members of class objects through pointers.
    if (r.GetHeight() * r.GetWidth() > i->GetHeight() * i->GetWidth())
        cout << " r ";
    else cout << " s ";
    cout << "has the greater area" << endl;
}
```

Program 2.3: A C++ code fragment demonstrating how *Rectangle* objects are declared and member functions invoked

2.1.4 Special Class Operations

Constructors and Destructors: The *constructor* and *destructor* are special member functions of a class. A constructor is a member function which initializes data members of an object. If a constructor is provided for a class, it is automatically executed when an object of that class is created. If a constructor is not defined for a class, memory is allocated for the data members of a class object, when it is created, but the data members are not initialized. The advantage of defining constructors for a class is that all class objects are well-defined as soon as they are created. This eliminates errors that result from accessing an undefined object. A *destructor* is a member function which deletes data members immediately before the object disappears. A constructor and destructor for class *Rectangle* are declared in Program 2.1.

A constructor must be declared as a public member function of its class; The name of a constructor must be identical to the name of the class to which it belongs; and a constructor must not specify a return type or return a value. Program 2.4 shows a constructor definition for class *Rectangle*.

```
Rectangle::Rectangle( int x, int y, int h, int w)
{
    xLow = x; yLow = y;
    height = h; width = w;
}
```

Program 2.4: Definition of a constructor for *Rectangle*

Constructors may be used to initialize *Rectangle* objects as follows:

```
Rectangle r (1, 3, 6, 6);
Rectangle *u = new Rectangle (0, 0, 3, 4);
```

These create *r*, a square of side 6 whose bottom left corner is at (1, 3); and *u*, a pointer to a *Rectangle* object of height 3 and width 4 whose bottom left corner is at the origin. Note that the declaration

```
Rectangle r;
```

will result in a compile time error. The reason is that the compiler requires a *default constructor* (a constructor with no arguments) to initialize *r*. Had we not defined the constructor of Program 2.4, the compiler would have generated its own default constructor. However, since we have defined a constructor, it is our responsibility to provide a default constructor if we wish to use the above declaration. Program 2.5 shows a definition of the *Rectangle* constructor that also serves as a default constructor. This is achieved by providing a default value for each argument in the argument list. In this case, the default for each argument is the integer 0. If this constructor is used, the above declaration for object *r* will result in the creation of a *Rectangle* object, all of whose data members are initialized to 0.

```
Rectangle::Rectangle( int x = 0, int y = 0, int h = 0, int w = 0)
: xLow (x), yLow (y), height (h), width (w)
{ }
```

Program 2.5: Sophisticated definition of a constructor for *Rectangle*

The constructor of Program 2.5 also differs from that of Program 2.4 in another respect: its body is empty; the data members are initialized by using a member initialization list (consisting of a colon followed by a list of data members and the arguments to which they are to be initialized in parentheses). Program 2.4 first initializes the data members and then assigns arguments to them in two separate steps, while Program 2.5 directly initializes the data members to the corresponding arguments in a single step. Thus, the latter approach results in a more efficient constructor.

Destructors are automatically invoked when a class object goes out of scope or when a class object is deleted. Like a constructor, a destructor must be declared as a public member of its class; its name must be identical to the name of its class prefixed with a tilde, ~; a destructor must not specify a return type or return a value, and a destructor may not take arguments. If a destructor is not defined for a class, the deletion of an object of that class results in the freeing of memory associated with data members of the class. If a data member is a pointer to some other object, the space allocated to the pointer is returned, but the object that it was pointing to is not deleted. If we also wish to delete this object, we must define a destructor that explicitly does so.

Operator Overloading: Consider the operator `==` which is used to check for equality between two data items. The operator `==` may be used to check for equality between two float data items; it can also be used to check for equality between two list items. The hardware algorithms implementing operator `==` depend on the type of the operands being compared; that is, the algorithm for comparing two floats is different from the one used to compare two ints. This is an example of operator overloading. However, if we were to try to use operator `==` to check for equality between two *Rectangle* objects, the compiler would complain that operator `==` is not defined for *Rectangle* objects. C++ allows the programmer to overload operators for user-defined data types. This is done by providing a definition that implements the operator for the particular data type. This definition takes the form of a class member function or an ordinary function, depending on the operator. The function prototype used must adhere to the specifications for the particular operator. For details about the specifications for various operators, see one of the introductory texts listed at the end of this chapter.

Program 2.6 overloads operator `==` for class *Rectangle*. Our program uses the this pointer which we describe briefly: The C++ keyword `this`, when used inside a member function of a class, represents a pointer to the particular class object that invoked it. The class object, itself, is therefore represented by `*this`.

We can now use the operator `==` to determine whether two rectangles are identical. Our program first evaluates the expression `"this == &r"`. This expression

```

bool Rectangle::operator==(const Rectangle& r)
{
    if (this == &r) return true;
    if ((xLow == r.xLow) && (yLow == r.yLow)
        && (height == r.height) && (width == r.width) ) return true;
    else return false;
}

```

Program 2.6: Overloading operator== for class *Rectangle*

checks to see if the two rectangles being compared are the same object. This would happen in the following two cases:

```

if (r == r) ...

Rectangle& u = r;
if (r == u) ...

```

In both cases, the expression "this == &r" evaluates to true because both operands represent the same object, and there is no need to compare the individual data members of the two rectangles. This is especially efficient if the class contains a large number of data members. If the expression evaluates to false, then the two rectangles are not the same object, and the individual data members must be compared. Program 2.7 overloads operator << so that *Rectangle* objects can be output by using cout.

```

ostream& operator<<(ostream& os, Rectangle& r)
{
    os << "Position is: " << r.xLow << " ";
    os << r.yLow << endl;
    os << "Height is: " << r.height << endl;
    os << "Width is: " << r.width << endl;
    return os;
}

```

Program 2.7: Overloading operator<< for class *Rectangle*

Notice that `operator<<` accesses private data members of class *Rectangle*. Therefore, it must be made a friend of *Rectangle*. In general, we want to minimize the number of friend declarations in a class because a friend represents an exception to the data encapsulation principle. However, there are instances, such as this, when they are necessary. The statement:

```
cout << r;
```

where *r* is defined as a square of side 6 with bottom left corner at (1, 3), will print the following:

```
Position is: 1 3
Height is: 6
Width is: 6
```

2.1.5 Miscellaneous Topics

In C++, a **struct** is identical to a class, except that the default level of access is public; that is, if the struct definition of a data type does not specify whether a given member (data or function) has public, private, or protected access, then the member has public access. In a class, the default is private access. Thus, the C++ struct is a generalization of the C struct.

A **union** is a structure that reserves storage for the largest of its data members so that only one of its data members can be stored, at any time. This is useful in applications where it is known that only one of many possible data items, each of a different type, needs to be stored in a structure; but there is no way to know what that data type is until runtime. The struct or class structures reserve memory for all their data members. Thus, using a union results in a more memory-efficient program, in these cases. We will use union in Chapter 4 on linked lists; we will also study a technique for improving on union by using inheritance.

A static class data member may be thought of as a global variable for its class. From the perspective of a class member function, a static data member is like any other data member. One difference is that each class object does not have its own exclusive copy. There is only one copy of a static data member and all class objects must share it. A second difference is that the declaration of a static data member in its class does not constitute a definition. Consequently, a definition of the data member is required somewhere else in the program. We

82 Arrays

will see an example of a static class member later in this chapter, when we implement the *Polynomial* data type.

2.1.6 ADTs and C++ classes

Example 2.1 (Abstract data type *NaturalNumber*): ADT 2.1 contains the class definition of *NaturalNumber*. You will notice that the class definition of *NaturalNumber* in ADT 2.1 is very similar to the ADT definition of ADT 1.1. As a result, we will henceforth use the C++ class to define an ADT instead of the notation of ADT 1.1. Note that there is one significant aspect in which the format of ADT 1.1 differs from the C++ class: some operators in C++ such as `operator++`, when overloaded for user-defined ADTs, do not exist as member functions of the corresponding class. Rather, these operators exist as ordinary C++ functions. Thus, these operations are declared outside the C++ class definition of the ADT even though they are actually part of the ADT.

```
class NaturalNumber {
// An ordered subrange of the integers starting at zero and ending at
// the maximum integer (MAXINT) on the computer.
public:
    NaturalNumber Zero() {
        // Returns 0.

    }

    bool IsZero() {
        // If *this is 0, return true; otherwise, return false.

    }

    NaturalNumber Add(NaturalNumber y) {
        // Return the smaller of *this + y and MAXINT.

    }

    bool Equal(NaturalNumber y) {
        // Return true if *this == y, otherwise return false.

    }

    NaturalNumber Successor() {
        // If *this is MAXINT return MAXINT; otherwise return *this + 1.

    }

    NaturalNumber Subtract(NaturalNumber y) {
        // If *this < y, return 0; otherwise return *this - y.

    }
};
```

ADT 2.1: Abstract data type *NaturalNumber*

EXERCISES

1. Overload operator< for class *Rectangle* such that $r < s$ if and only if the area of r is less than that of s .
2. Write and test C++ code for the class *MyRectangle*, which is an enhanced version of the class *Rectangle* (Program 2.1). In addition to the data members listed in Program 2.1, *MyRectangle* has the data member *color*. You must include functions to change as well as to return the value of each data member and functions to return the area and perimeter of a rectangle. The operators << and >> should be overloaded to work with rectangles.
3. Write and test code for the C++ class *Currency*, which represents currency objects. Each currency object has two data members *\$* and *cents*, where *cents* is an integer between 0 and 99. You must include functions to set and return the data members, add and subtract currency objects, to multiply a currency object by numbers such as 2, 2.5 and so on. The operators << and >> should be overloaded to work with currency objects.
4. Implement a class *Complex*, which represents the Complex Number data type. Implement the following operations:
 - (a) A constructor (including a default constructor which creates the complex number $0 + 0i$).
 - (b) Overload operator+ to add two complex numbers.
 - (c) Overload operator* to multiply two complex numbers.
 - (d) Overload << and >> to print and read complex numbers. To do this, you will need to decide what you want your input and output format to look like.

Write a program according to the following specifications: use the constructor to define two complex numbers: $3 + 2i$ and $0 + 0i$. Input two complex numbers $5 + 3i$ and $0 + 0i$ using cin. Obtain the sum and product of all four complex numbers using operators + and *, respectively. Output the results using cout.

5. Implement a class *Quadratic* that represents 2-degree polynomials i.e., polynomials of type $ax^2 + bx + c$. Your class will require three data members corresponding to a , b , and c . Implement the following operations:
 - (a) A constructor (including a default constructor which creates the 0 polynomial).
 - (b) Overload operator+ to add two polynomials of degree 2.
 - (c) Overload << and >> to print and read polynomials. To do this, you will need to decide what you want your input and output format to look like.

84 Arrays

- (d) A function `Eval` that computes the value of a polynomial for a given value of x .
- (e) A function that computes the two solutions of the equation $ax^2+bx+c=0$. [Hint: use class `Complex` from the previous exercise.]

2.2 THE ARRAY AS AN ABSTRACT DATA TYPE

We begin our discussion by considering an array as an ADT. This is not the usual perspective, since many programmers view an array only as a consecutive set of memory locations. This is unfortunate because it clearly shows an emphasis on implementation issues. Although an array is usually implemented as a consecutive set of memory locations, this is not always the case. Intuitively, an array is a set of pairs, $\langle \text{index}, \text{value} \rangle$, such that each index that is defined has a value associated with it. In mathematical terms, we call this a *correspondence* or a *mapping*. However, when considering an ADT, we are more concerned with the operations that can be performed on an array. Aside from creating a new array, most languages provide only two standard operations for arrays, one that retrieves a value and one that stores a value. ADT 2.2 shows a class definition of the array ADT.

The constructor `GeneralArray(int j, RangeList list, float initValue = defaultValue)` produces a new array of the appropriate size and type. All of the items are initially set to the floating point variable `initValue`. `Retrieve` accepts an index and returns the value associated with the index if the index is valid or an error if the index is invalid. `Store` accepts an index, and a value of type float, and replaces the $\langle \text{index}, \text{oldvalue} \rangle$ pair with the $\langle \text{index}, \text{newvalue} \rangle$ pair.

`GeneralArray` is more general than the C++ array as it is more flexible about the composition of the index set. The C++ array requires the index set to be a set of consecutive integers starting at 0. Also, C++ does not check an array index to ensure that it belongs to the range for which the array is defined. Because of these disadvantages, it is often beneficial to define a more general and robust array class. Some of the member functions required to implement such a class are examined in the exercises. A general array class would typically be internally implemented by using the C++ array. To do this, it is necessary to be able to map a position in the general array onto a position in the C++ array. This is discussed in Section 2.5.

To simplify the discussion, we will hereafter use the C++ array. Therefore, before continuing, let us briefly review the C++ array. A C++ array `floatArray` of floats with index set ranging from 0 to $n-1$ is defined by:

```
float floatArray[n];
```

The i th element of `floatArray` may be accessed in two ways: `floatArray[i]` and `*(floatArray + i)`. In the latter mechanism, the variable `floatArray` is actually

```

class GeneralArray {
// A set of pairs <index, value> where for each value of index in IndexSet
// there is a value of type float. IndexSet is a finite ordered set of one or more
// dimensions, for example, {0, ..., n-1} for one dimension, {(0, 0),
// (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)} for two dimensions, etc.
public:
    GeneralArray(int j, RangeList list, float initValue = defaultValue);
// This constructor creates a j dimensional array of floats; the
// range of the kth dimension is given by the kth element of list. For each
// index i in the index set, insert <i, initValue> into the array.

    float Retrieve(index i);
// If i is in the index set of the array, return the float associated with i
// in the array; otherwise throw an exception.

    void Store(index i, float x);
// If i is in the index set of the array, replace the old value associated with i
// by x; otherwise throw an exception.
}; //

```

ADT 2.2: Abstract data type *GeneralArray*

a pointer to the zeroth element of the array. The expression *floatArray + i* is a pointer to the *i*th element of array *floatArray*; it follows that **(floatArray + i)* is the *i*th element of array *floatArray*.

EXERCISES

1. Implement a class *CppArray* which is identical to a one-dimensional C++ array (i.e., the index set is a set of consecutive integers starting at 0) except for the following:
 - (i) It performs range checking.
 - (ii) It allows one array to be assigned to another array through the use of the assignment operator (e.g., *cp1 = cp2*).
 - (iii) It supports a function that returns the size of the array.
 - (iv) It allows the reading or printing of arrays through the use of *cin* and *cout*.

To do this, you will have to define the following member functions on *CppArray*:

86 Arrays

- (a) A constructor `CppArray (int size = defaultSize, float initvalue = defaultValue)`. This creates an array of size `size` all of whose elements are initialized to `initvalue`.
- (b) A copy constructor `CppArray (const CppArray& cp2)`. This creates an array identical to `cp2`. Copy constructors are used to initialize a class object with another class object as in the following:
`CppArray a = b ;`
- (c) An assignment operator `CppArray& operator= (const CppArray& cp2)`. This replaces the original array with `cp2`.
- (d) A destructor `~CppArray()`.
- (e) The subscript operator `float& operator[] (int i)`. This should be implemented so that it performs range-checking.
- (f) A member function `int GetSize()`. This returns the size of the array.
- (g) Implement functions to read and print the elements of a `CppArray` by overloading operators `<<` and `>>`. These functions will have to be made friends of `CppArray`.

2.3 THE POLYNOMIAL ABSTRACT DATA TYPE

Arrays are not only data structures in their own right; we can also use them to implement other abstract data types. For instance, let us consider one of the simplest and most common data structures: the *ordered*, or *linear*, list. We can find many examples of this data structure, including:

- Days of the week: (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday)
- Values in a deck of cards: (Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King)
- Floors of a building: (basement, lobby, mezzanine, first, second)
- Years the United States fought in World War II: (1941, 1942, 1943, 1944, 1945)
- Years Switzerland fought in World War II: ()

Notice that “Years Switzerland fought in World War II” is different because it contains no items. It is an example of an empty list, which we denote as $()$. The other lists all contain items that are written in the form $(a_0, a_1, \dots, a_{n-1})$.

We can perform many operations on lists, including:

- (1) Find the length, n , of the list.

- (2) Read the list from left to right (or right to left).
- (3) Retrieve the i th element, $0 \leq i < n$.
- (4) Store a new value into the i th position, $0 \leq i < n$.
- (5) Insert a new element at the position i , $0 \leq i < n$, causing elements numbered $i, i + 1, \dots, n - 1$ to become numbered $i + 1, i + 2, \dots, n$.
- (6) Delete the element at position i , $0 \leq i < n$, causing elements numbered $i + 1, \dots, n - 1$ to become numbered $i, i + 1, \dots, n - 2$.

It is not always necessary to be able to perform all of these operations; often a subset will suffice. In the study of data structures we are interested in ways of representing ordered lists so that these operations can be carried out efficiently.

Rather than state the formal specification of the ADT *ordered list*, we want to explore briefly its implementation. Perhaps the most common way to represent an ordered list is by an array where we associate the list element a_i with the array index i . We will refer to this as *sequential mapping* because, using the conventional array representation, we are storing a_i and a_{i+1} into consecutive locations i and $i + 1$ of the array. This gives us the ability to retrieve or modify the values of random elements in the list in a constant amount of time, essentially because a computer has random access to any word in its memory. We can access the list element values in either direction by changing the subscript values in a controlled way. Only operations (5) and (6) require real effort. Insertion and deletion using sequential allocation force us to move some of the remaining elements so that the sequential mapping is preserved in its proper form. It is precisely this overhead that leads us to consider nonsequential mappings of ordered lists in Chapter 4.

Let us jump right into a problem requiring ordered lists, which we shall solve by using one-dimensional arrays. The problem calls for building an ADT for the representation and manipulation of polynomials in a single variable (say x). Two such polynomials are

$$a(x) = 3x^2 + 2x - 4 \quad \text{and} \quad b(x) = x^3 - 10x^2 - 3x^2 + 1$$

The polynomial $a(x)$ has 3 terms $3x^2$, $2x$, and -4 . The coefficients of these terms are 3, 2, and -4 , respectively; their exponents are 2, 1, and 0. A term of a polynomial may be represented as a (coefficient, exponent) pair. For example, (3, 2) represents the term $3x^2$. A term whose coefficient is nonzero is called a *nonzero term*. Normally, terms with zero coefficients are not displayed. The term with exponent equal to zero (e.g., $(-4, 0)$ in $a(x)$) does not show the variable, since x raised to a power of zero is 1. The *degree* of a polynomial is the largest exponent from among the nonzero terms. There are standard mathematical definitions for the sum and product of polynomials. Assume that we have two polynomials,

88 Arrays

$a(x) = \sum a_i x^i$ and $b(x) = \sum b_i x^i$; then

$$a(x) + b(x) = \sum (a_i + b_i) x^i$$

$$a(x) \cdot b(x) = \sum (a_i x^i \cdot \sum (b_j x^j))$$

Similarly, we can define subtraction and division on polynomials, as well as many other operations. We begin with an ADT definition (ADT 2.3) of a polynomial. ADT 2.3 shows only a subset of the functions we may want to perform on polynomials. The ADT is easily extended to include functions for the subtract, divide, input and output operations.

```
class Polynomial {
//  $p(x) = a_n x^{e_n} + \dots + a_0 x^{e_0}$ ; a set of ordered pairs of  $\langle a_i, e_i \rangle$ ,
// where  $a_i$  is a nonzero float coefficient and  $e_i$  is a non-negative integer exponent.
public:
    Polynomial();
    // Construct the polynomial  $p(x) = 0$ .

    Polynomial Add(Polynomial poly);
    // Return the sum of the polynomials *this and poly.

    Polynomial Mult(Polynomial poly);
    // Return the product of the polynomials *this and poly.

    float Eval(float f);
    // Evaluate the polynomial *this at  $f$  and return the result.
};
```

ADT 2.3: Abstract data type *Polynomial*

2.3.1 Polynomial Representation

We are now ready to make some representation decisions. A very reasonable first decision would be to arrange the terms in decreasing order of exponent. This considerably simplifies many of the operations. We discuss three representations that are based on this principle:

Representation 1: One way to represent polynomials in C++ is to define the private data members of *Polynomial* as follows:

```
private:
    int degree;           // degree  $\leq$  MaxDegree
    float coef [MaxDegree + 1]; // coefficient array
```

where *MaxDegree* is a constant that represents the largest-degree polynomial that is to be represented. Now, if *a* is a *Polynomial* class object and $n \leq \text{MaxDegree}$, then the polynomial $a(x)$ above would be represented as:

```
a.degree = n
a.coef[i] =  $a_{n-i}$ ,  $0 \leq i \leq n$ 
```

Note that *a.coef[i]* is the coefficient of x^{n-i} and the coefficients are stored in order of decreasing exponents. This representation leads to very simple algorithms for many of the operations on polynomials (addition, subtraction, evaluation, multiplication, etc.).

Representation 2: Representation 1 requires us to know the maximum degree of the polynomials we expect to work with and also is quite wasteful in its use of computer memory. For instance, if *a.degree* is much less than *MaxDegree*, then most of the positions in the array *a.coef*[] are unused. We can overcome both of these deficiencies by defining *coef* so that its size is *a.degree* + 1. This can be done by declaring the following private data members:

```
private:
    int degree;
    float *coef;
```

and adding the following constructor to *Polynomial*:

```
Polynomial::Polynomial(int d)
{
    degree = d;
    coef = new float [degree+1];
}
```

Representation 3: Although Representation 2 solves the problems mentioned earlier, it does not yield a desirable representation. To see this, let us consider polynomials that have many zero terms. Such polynomials are called *sparse*. For instance, the polynomial $x^{1000} + 1$ has two nonzero terms and 999 zero terms. Consequently, 999 of the entries in *coef* will be zero if Representation 2 is used. To overcome this problem, we store only the nonzero terms. For this purpose, we define the class *term* as below.

90 Arrays

class *Polynomial*; // forward declaration

```
class Term {  
    friend Polynomial;  
private:  
    float coef; // coefficient  
    int exp;    // exponent  
};
```

The private data members of *Polynomial* are defined as follows:

```
private:  
    Term *termArray; // array of nonzero terms  
    int capacity;     // size of termArray  
    int terms;        // number of nonzero terms
```

Before proceeding, we should compare our current representation with Representation 2. Representation 3 is definitely superior when there are many zero terms. For example, $c(x) = 2x^{1000} + 1$ uses only 6 units of space (one for $c.capacity$, one for $c.terms$, two for the coefficients, and two for the exponents) when we use Representation 3 but when Representation 2 is used, 1002 units of space are required. However, when all terms are nonzero, as in $a(x)$ above, Representation 3 uses about twice as much space as does Representation 2. Unless we know beforehand that each of our polynomials has very few zero terms, Representation 3 is preferable. The exercises explore an alternative representation that uses the STL class *vector* instead of an array.

2.3.2 Polynomial Addition

Let us now write a C++ function to add two polynomials, a and b , to obtain the sum $c = a + b$. It is assumed that Representation 3, above, is used to store a and b . Function *Add* (Program 2.8) adds $a(x)$ ($*this$) and $b(x)$ term by term to produce $c(x)$. This function assumes that the default constructor for *Polynomial* initializes *capacity* and *terms* to 1 and 0, respectively, and initializes *termArray* to an array with 1 position (i.e., the size or capacity of *termArray* is 1).

The basic loop of this algorithm consists of merging the terms of the two polynomials, depending upon the result of comparing the exponents. The *if* statement determines how the exponents are related and performs the proper action. Since the tests within the *while* statement require two terms, if one polynomial runs out of terms, we must exit the loop. The remaining terms of the other polynomial can be copied directly into the result. The terms of c are entered into its array *termArray* by calling function *NewTerm* (Program 2.9). In

```

1  Polynomial Polynomial::Add(Polynomial b)
2  { // Return the sum of the polynomials *this and b.
3      Polynomial c;
4      int aPos = 0, bPos = 0;
5      while ((aPos < terms) && (bPos < b.terms))
6          if ((termArray[aPos].exp == b.termArray[bPos].exp) {
7              float t = termArray[aPos].coef + b.termArray[bPos].coef;
8              if (t) c.NewTerm(t, termArray[aPos].exp);
9              aPos++; bPos++;
10         }
11         else if ((termArray[aPos].exp < b.termArray[bPos].exp) {
12             c.NewTerm(b.termArray[bPos].coef, b.termArray[bPos].exp);
13             bPos++;
14         }
15         else {
16             c.NewTerm(termArray[aPos].coef, termArray[aPos].exp);
17             aPos++;
18         }
19         // add in remaining terms of *this
20         for (; aPos < terms; aPos++)
21             c.NewTerm(termArray[aPos].coef, termArray[aPos].exp);
22         // add in remaining terms of b(x)
23         for (; bPos < b.terms; bPos++)
24             c.NewTerm(b.termArray[bPos].coef, b.termArray[bPos].exp);
25         return c;
26     }

```

Program 2.8: Adding two polynomials

case there is not enough space in *termArray* to accommodate the new term, its capacity is doubled. If we don't have enough memory to create the array *temp*, new will throw an exception. Since neither *NewTerm* nor *Add* catch this exception, control is passed to the function that invoked *Add* whenever new fails. If the thrown exception is not caught by any part of the program, the program terminates.

Analysis of Add: It is natural to carry out this analysis in terms of the number of nonzero terms in *a* (*this) and *b*. Let *m* and *n* be the number of nonzero terms in *a* and *b* respectively. Lines 3 and 4 contribute $O(1)$ to the overall computing time. In each iteration of the **while** loop, *aPos* or *bPos* or both increase by 1.

9.2 Arrays

```

void Poly(lexical: NewTerm(const float theCoeff, const int theExp)
// Add a new term to the end of termArray.
    if (terms == capacity)
        // double capacity of termArray
        capacity *= 2;
        term *temp = new Term [capacity]; // new array
        copy(termArray, termArray + terms, temp);
        delete [] termArray; // deallocate old memory
        termArray = temp;
    }
    termArray[terms].coeff = theCoeff;
    termArray[terms++].exp = theExp;
}

```

Program 2.9: Adding a new term, doubling array size when necessary

Since the **while** loop terminates when either *aPos* equals *a.terms* or *bPos* equals *b.terms*, the number of iterations of this loop is bounded by $m + n - 1$. This worst-case is achieved, for instance, when $a(x) = \sum_{i=0}^m x^{2i}$ and $b(x) = \sum_{i=0}^n x^{2i+1}$. Since none of the exponents are the same in $a(x)$ and $b(x)$, *termArray*[*aPos*].*exp* \neq *termArray*[*bPos*].*exp*. Consequently, on each iteration the value of only one of *aPos* or *bPos* increases by 1. For the moment, let's ignore the time spent doubling the capacity of *c.termArray* in *NewTerm*. Exclusive of this time, each iteration of the **while** loop takes $O(1)$ time. So, the total time contributed by this loop is $O(m + n)$. The **for** loops of lines 20 and 23 also contribute $O(m + n)$ time to the overall complexity. Adding together the time contributions of all components of *Add*, we obtain $O(m + n + \text{time spent in array doubling})$ as the asymptotic computing time of this algorithm.

Although it may appear that a lot of time is spent doubling the size of *termArray*, this is actually not the case. To create an array of type *Term* and size *s*, we must allocate sufficient space for *s* terms and also invoke the constructor for *Term* for each position in the array. Following the creation of the new array, we copy elements from the old array into the new one. In all our analyses of array doubling, we assume that it takes $O(1)$ time to allocate memory and that each invocation of the constructor and each element copy takes $O(1)$ time. Hence the time needed to double an array is linear in the size of the new array. Initially, *c.capacity* is 1. Suppose that when *Add* terminates, *c.capacity* is 2^k for some k , $k > 0$. The total time spent over all array doublings is $O(\sum_{i=1}^k 2^i) = O(2^{k+1}) = O(2^k)$.

Since $c.terms > 2^{k-1}$ (otherwise the array size would not have been doubled from 2^{k-1} to 2^k) and $m + n \geq c.terms$, the total time spent in array doubling is $O(c.terms) = O(m + n)$. Hence even with the time spent on array doubling added in, the total run time of *Add* is $O(m + n)$.

Array doubling contributes at most a constant multiplicative factor to the run time of *Add*! This latter statement is true even if we resize using any constant multiplicative factor greater than 1 (rather than by a factor of 2 as in array doubling) but is not true when we resize by an additive constant (i.e., increase the size by c , $c \geq 1$, whenever additional space is needed). Experiments indicate that array doubling is responsible for a very small fraction of the total run time of *Add*. \square

Notice that when *Add* terminates, capacity $2.terms$ for the result polynomial. We can recover any excess space in the array *termArray* by reducing its size to equal the number of terms in the polynomial. The code to reduce the size of an array is very similar to that used in Program 2.9 to increase array size. This reduction in size doesn't change the asymptotic complexity of *Add*.

The exercises examine an alternative array representation of polynomials and, in Chapter 4, we develop a linked representation for them.

EXERCISES

1. Use the six operations defined in this section for an ordered list to arrive at an ADT specification for such a list.
2. If $a = \{a_0, \dots, a_{k-1}\}$ and $b = \{b_0, \dots, b_{m-1}\}$ are ordered lists, then $a < b$ if $a_i = b_i$ for $0 \leq i < j$ and $a_j < b_j$, or, if $a_i = b_i$ for $0 \leq i < n$ and $n < m$. Write a function that returns -1 , 0 , or $+1$, depending upon whether $a < b$, $a = b$, or $a > b$. Assume that the a_i s and b_j s are integer.
3. Modify function *Add* so that it reduces the size of *c.termArray* to *c.terms* prior to termination. With this modification, can we dispense with the data member *capacity*?
4. Write C++ functions to input and output polynomials represented as in this section. Your functions should overload the $<<$ and $>>$ operators.
5. Write a C++ function that multiplies two polynomials represented as in this section. What is the computing time of your function?
6. Write a C++ function that evaluates a polynomial at a value x_0 using the representation of this section. Try to minimize the number of operations.
7. The polynomials $a(x) = x^{2n} + x^{2n-2} + \dots + x^2 + x^0$ and $b(x) = x^{2n+1} + x^{2n-1} + \dots + x^1 + x$ cause *Add* to work very hard. For these polynomials, determine the exact number of times each statement will be executed.

8. Develop a C++ class *Polynomial* that uses an STL vector instead of an array to hold the nonzero terms. You must include functions to add, subtract and multiply polynomials. Also, you must overload the input and output operators << and >> so that these work with objects of type *Polynomial*. What is the complexity of each function in your class? Compare the merits of using a vector over using an array.
9. In an alternative representation of polynomials, we store the nonzero terms of all polynomials in a single array called *termArray*. Each element in *termArray* is of type *Term*, which we defined in the section. Since the single array *termArray* is to be shared by all *Polynomial* objects, we declare it as a static class data member of *Polynomial*. The private data members of *Polynomial* are defined as follows:

```
private:
    static Term *termArray;
    static int capacity;
    static int free;
    int start, finish;
```

where *capacity* is the size of the array *termArray*. The required definitions of the static class members outside the class definition are:

```
int Polynomial::capacity = 100;
Term Polynomial::termArray = new Term[100];
int Polynomial::free = 0; // location of next free location in termArray
```

Consider the two polynomials $a(x) = 2x^{1000} + 1$ and $b(x) = x^4 + 10x^3 + 3x^2 + 1$. These could be stored in the array *termArray* as shown in Figure 2.1. Note that *a.start* and *b.start* give the location of the first term of *a* and *b* respectively, whereas *a.finish* and *b.finish* give the location of the last term of *a* and *b*. The static class member *free* gives the location of the next free location in the array *termArray*. For our example, *a.start* = 0, *a.finish* = 1, *b.start* = 2, *b.finish* = 5, and *free* = 6.

In general, any polynomial *a* that has *n* nonzero terms has *a.start* and *a.finish* such that *a.finish* = *a.start* + *n* - 1. If *a* has no nonzero terms (i.e., *a* is the zero polynomial), then *a.finish* = *a.start* - 1.

Write and test C++ functions to input, output, add, multiply, and evaluate polynomials represented in this way. Use array doubling to increase the size of *termArray* whenever needed. Is the representation of this exercise better or worse than the representation used in the text?

	<i>a.start</i>	<i>a.finish</i>	<i>b.start</i>		<i>b.finish</i>	<i>free</i>
	↓	↓	↓		↓	↓
<i>coef</i>	2	1	1	10	3	1
<i>exp</i>	1000	0	4	3	2	0
	0	1	2	3	4	5
						6

Figure 2.1: Array representation of two polynomials

10. Show that if we start with an array whose size is 10 and increase its size by a constant multiplicative factor c , $c > 1$, whenever we need additional space, the total time spent in all array resizings is linear in the final size of the array. (Assume that the size of the new array is obtained by rounding up the product of c and the initial array size, if this quantity is not a whole number.)
11. Show that if we start with an array whose size is 1 and increase its size by a constant additive factor c , $c \geq 1$, whenever we need additional space, the total time spent in all array resizings is quadratic in the final size of the array.

2.4 SPARSE MATRICES

2.4.1 Introduction

A matrix is a mathematical object that arises in many physical problems. As computer scientists, we are interested in studying ways to represent matrices so that the operations to be performed on them can be carried out efficiently. A general matrix consists of m rows and n columns of numbers, as in Figure 2.2. The first matrix has five rows and three columns, the second six rows and six columns. In general, we write $m \times n$ (read “ m by n ”) to designate a matrix with m rows and n columns. Such a matrix has mn elements. When m is equal to n , we call the matrix square.

It is very natural to store a matrix in a two-dimensional array, say $a[m][n]$. Then we can work with any element by writing $a[i][j]$, and this element can be

	col 0	col 1	col 2		col 0	col 1	col 2	col 3	col 4	col 5
row 0	-27	3	4	row 0	15	0	0	22	0	-15
row 1	6	82	-2	row 1	0	11	3	0	0	0
row 2	109	-64	11	row 2	0	0	0	-6	0	0
row 3	12	8	9	row 3	0	0	0	0	0	0
row 4	48	27	47	row 4	91	0	0	0	0	0
				row 5	0	0	28	0	0	0

(a)

(b)

Figure 2.2: Two matrices

found very quickly, as we will see in the next section. Now if we look at the second matrix of Figure 2.2, we see that it has many zero entries. Such a matrix is called *sparse*. There is no precise definition of when a matrix is sparse and when it is not, but it is a concept that we can all recognize intuitively. In the matrix of Figure 2.2(b), only eight out of 36 possible elements are nonzero, and that is sparse! A sparse matrix requires us to consider an alternative form of representation. This comes about because in practice many of the matrices we want to deal with are large, e.g., 5000×5000 , but at the same time they are sparse: say only 5000 out of 25 million possible elements are nonzero. When a 5000×5000 array is used to store this matrix, we need 25 million units of space. Also, 25 million units of time are needed for operations such as addition and transposition. Using an alternative representation, which stores only the nonzero elements, we can reduce both the space and time requirements considerably.

Before developing a particular representation for sparse matrices, we first must consider the operations that we want to perform on these matrices. A minimal set of operations includes matrix creation, transposition, addition, and multiplication. ADT 2.4 contains our specification of the matrix ADT.

```

class SparseMatrix
// A set of triples, <row, column, value>, where row and column are non-negative
// integers and form a unique combination; value is also an integer.
public:
    SparseMatrix(int r, int c, int t);
    // The constructor function creates a SparseMatrix with
    // r rows, c columns, and a capacity of t nonzero terms.

    SparseMatrix Transpose();
    // Returns the SparseMatrix obtained by interchanging the row and column
    // value of every triple in *this.

    SparseMatrix Add(SparseMatrix b);
    // If the dimensions of *this and b are the same, then the matrix produced by
    // adding corresponding items, namely those with identical row and column
    // values is returned; otherwise, an exception is thrown.

    SparseMatrix Multiply(SparseMatrix b);
    // If the number of columns in *this equals the number of rows in b then the
    // matrix d produced by multiplying *this and b according to the formula
    //  $d[i][j] = \sum_k (a[i][k] \cdot b[k][j])$ , where  $d[i][j]$  is the  $(i, j)$ th element, is returned
    // k ranges from 0 to one less than the number of columns in *this.
    // otherwise, an exception is thrown.
};

```

ADT 2.4: Abstract data type *SparseMatrix*

1.4.2 Sparse Matrix Representation

Before implementing any of these operations, we must establish the representation of the sparse matrix. By examining Figure 2.2, we know that we can characterize uniquely any element within a matrix by using the triple <row, col, value>. This means that we can use an array of triples to represent a sparse matrix. We require that these triples be stored by rows with the triples for the first row first, followed by those of the second row, and so on. We also require that all the triples for any row be stored so that the column indices are in ascending order. In addition, to ensure that the operations terminate, we must know the number of rows and columns and the number of nonzero elements in the matrix. Fusing all this information together suggests that we define:

98 Arrays

class *SparseMatrix* ; // forward declaration

```
class MatrixTerm {
friend class SparseMatrix;
private:
    int row, col, value;
};
```

and in class *SparseMatrix*:

```
private:
    int rows, cols, terms, capacity;
    MatrixTerm *smArray;
```

where *rows* is the number of rows in the matrix; *cols* is the number of columns; *terms* is the total number of nonzero entries; and *capacity* is the size of *smArray*. Figure 2.3(a) shows the representation of the matrix of Figure 2.2(b) using *smArray*. Positions 0 through 7 of *smArray* store the triples representing the nonzero entries. The triples are ordered by row and within rows by columns.

	row	col	value		row	col	value
<i>smArray</i> [0]	0	0	15	<i>smArray</i> [0]	0	0	15
[1]	0	3	22	[1]	0	4	91
[2]	0	5	-15	[2]	1	1	11
[3]	1	1	11	[3]	2	1	3
[4]	1	2	3	[4]	2	5	28
[5]	2	3	-6	[5]	3	0	22
[6]	4	0	91	[6]	3	2	-6
[7]	5	2	28	[7]	5	0	-15
(a)				(b)			

Figure 2.3: Sparse matrix and its transpose stored as triples

2.4.3 Transposing a Matrix

Figure 2.3(b) shows the transpose of the matrix of Figure 2.3(a). To transpose a matrix, we must interchange the rows and columns. This means that if an element is at position $[i][j]$ in the original matrix, then it is at position $[j][i]$ in the transposed matrix. When $i = j$, the elements on the diagonal will remain unchanged. Since the original matrix is organized by rows, our first idea for a transpose algorithm might be the following:

```
for (each row  $i$ )
    store  $(i, j, \text{value})$  of the original matrix as  $(j, i, \text{value})$  of the transpose;
```

The difficulty is in not knowing where to put the element (j, i, value) until all other elements that precede it have been processed. In Figure 2.3(a), for instance, we encounter

(0, 0, 15)	which becomes	(0, 0, 15)
(0, 3, 22)	which becomes	(3, 0, 22)
(0, 5, -15)	which becomes	(5, 0, -15)
(1, 1, 11)	which becomes	(1, 1, 11)

We can avoid this difficulty of not knowing where to place an element by changing the order in which we place elements into the transpose. Consider the following strategy.

```
for (all elements in column  $j$ )
    store  $(j, i, \text{value})$  of the original matrix as  $(j, i, \text{value})$  of the transpose;
```

This says “find all elements in column 0 and store them in row 0, find all elements in column 1 and store them in row 1, etc.” Since the rows are originally in order, this means that we will locate elements in the correct column order as well. The function *Transpose* (Program 2.10) computes and returns the transpose of **this*.

It is not too difficult to see that the function is correct. The variable *currentB* always gives us the position in *b* where the next term in the transpose is to be inserted. The terms in *b.mArray* are generated by rows. Since the rows of *b* are the columns of **this*, row *c* of *b* is obtained by collecting all the nonzero terms in column *c* of **this*. This is precisely what is being done in lines 7 through 15. In the first iteration of the for loop of lines 7 through 15, all terms from column 0 of **this* are moved to *b*; in the next iteration, all terms from column 1 are moved; and so on.

Analysis of Transpose: Let *cols*, *rows*, and *terms* denote the number of

```

1 SparseMatrix SparseMatrix::Transpose()
2 // Return the transpose of *this.
3 SparseMatrix b(cols, rows, terms); // capacity of b.smArray is terms
4 if (terms > 0)
5     // nonzero matrix
6     int currentB = 0;
7     for (int c = 0; c < cols; c++) // transpose by columns
8         for (int i = 0; i < rows; i++)
9             // find and move terms in column c
10             if (smArray[i].col == c)
11             {
12                 b.smArray[currentB].row = c;
13                 b.smArray[currentB].col = smArray[i].row;
14                 b.smArray[currentB++].value = smArray[i].value;
15             }
16     // end of if (terms > 0)
17     return b;
18 }

```

Program 2.10: Transposing a matrix

column, row, and term, respectively, in **this*. For each iteration of the loop of lines 7 through 15, the condition of line 10 is tested *terms* times. Since the number of iterations of this loop is *cols*, the total time for line 10 is *terms* · *cols*. The assignments of lines 12-14 take place exactly *terms* times, since there are only this many nonzero terms in the sparse matrix being generated. Lines 3-6 take a constant amount of time. The total time for *Transpose* is therefore $O(\text{terms} \cdot \text{cols})$. In addition to the space needed for **this* and *b*, the function requires only a constant amount of space, i.e., space for the variables *c*, *i*, and *currentB*. □

We now have a matrix transpose algorithm that we believe is correct and that has a computing time of $O(\text{terms} \cdot \text{cols})$. This computing time is a little disturbing, since we know that in case the matrices had been represented as two-dimensional arrays, we could have obtained the transpose of a *rows* × *cols* matrix in time $O(\text{rows} \cdot \text{cols})$. The algorithm for this has the simple form

```

for (int j = 0; j < cols; j++)
    for (int i = 0; i < rows; i++)
        b[j][i] = a[i][j];

```

The $O(\text{terms} \cdot \text{cols})$ time for function *transpose* becomes $O(\text{rows} \cdot \text{cols}^2)$ when *terms* is of the order of $\text{rows} \cdot \text{cols}$. This is worse than the $O(\text{rows} \cdot \text{cols})$ time using the simple form. Perhaps, in an effort to conserve space, we have traded away too much time. Actually, we can transpose a matrix represented as a sequence of triples in time $O(\text{terms} + \text{cols})$ by using a little more space. The new algorithm, *FastTranspose* (Program 2.11), proceeds by first determining the number of elements in each column of **this*. This gives us the number of elements in each row of *b*. From this information, the starting point in *b* of each of its rows is easily obtained. We can now move the elements of **this* one by one into their correct position in *b*.

The correctness of function *FastTranspose* follows from the preceding discussion and the observation that the starting point, *rowStart*[*i*], of row *i*, $i > 0$, of *b* is *rowStart*[*i* - 1] + *rowSize*[*i* - 1], where *rowSize*[*i* - 1] is the number of elements in row *i* - 1 of *b*. The computation of *rowSize* and *rowStart* is carried out in lines 9-13. In lines 14-21 the elements of **this* are placed one by one into the correct place in *b*. *rowStart*[*j*] is maintained so that it is always the position in *b* where the next element in row *j* of *b* is to be inserted. If we try the algorithm on the sparse matrix of Figure 2.3(a), then after the execution of line 13, the values of *rowSize* and *rowStart* are:

	[0]	[1]	[2]	[3]	[4]	[5]
<i>rowSize</i> =	3	2	1	0	1	1
<i>rowStart</i> =	0	3	5	6	6	7

There are three loops in *FastTranspose*, which are executed *terms*, *cols* - 1, and *terms* times respectively. Each iteration of each loop takes a constant amount of time, so the time for all 3 loops is $O(\text{cols} + \text{terms})$. Line 9 takes $O(\text{cols})$ time and the remaining lines take $O(1)$ time. So, the overall complexity is $O(\text{cols} + \text{terms})$.

The computing time of $O(\text{cols} + \text{terms})$ becomes $O(\text{rows} \cdot \text{cols})$ when *terms* is of the order of $\text{rows} \cdot \text{cols}$. This is the same as when two-dimensional arrays were in use. However, the constant factor associated with *FastTranspose* is larger than that for the array algorithm. When *terms* is sufficiently small compared to its maximum of $\text{rows} \cdot \text{cols}$, *FastTranspose* will be faster. Hence in this representation, we save both space and time! This was not true of *Transpose*, since *terms* will almost always be greater than $\max\{\text{rows}, \text{cols}\}$ and $\text{cols} \cdot \text{terms}$ will therefore always be at least $\text{rows} \cdot \text{cols}$. The constant factor associated with *Transpose* is also larger than the one in the array algorithm. Finally, you should note that *FastTranspose* requires more space than does *Transpose*. The space required by *FastTranspose* can be reduced by utilizing the same space to represent the two arrays *rowSize* and *rowStart*.

```

1 SparseMatrix SparseMatrix::FastTranspose()
2 { // Return the transpose of *this in O(terms + cols) time.
3   SparseMatrix b(cols, rows, terms);
4   if (terms > 0)
5   { // nonzero matrix
6     int *rowSize = new int(cols);
7     int *rowStart = new int(cols);
8     // compute rowSize[i] = number of terms in row i of b
9     fill(rowSize, rowSize + cols, 0); // initialize
10    for (i = 0; i < terms; i++) rowSize[smArray[i].col]++;

11    // rowStart[i] = starting position of row i in b
12    rowStart[0] = 0;
13    for (i = 1; i < cols; i++) rowStart[i] = rowStart[i - 1] + rowSize[i - 1];

14    for (i = 0; i < terms; i++)
15    { // copy from *this to b
16      int j = rowStart[smArray[i].col];
17      b.smArray[j].row = smArray[i].col;
18      b.smArray[j].col = smArray[i].row;
19      b.smArray[j].value = smArray[i].value;
20      rowStart[smArray[i].col]++;
21    } // end of for
22    delete [] rowSize;
23    delete [] rowStart;
24  } // end of if
25  return b;
26 }

```

Program 2.11: Transposing a matrix faster

2.4.4 Matrix Multiplication

Definition: Given a and b , where a is $m \times n$ and b is $n \times p$, the product matrix d has dimension $m \times p$. Its $[i][j]$ element is

$$d_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj}$$

for $0 \leq i < m$ and $0 \leq j < p$. \square

The product of two sparse matrices may no longer be sparse, as Figure 2.4

shows.

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Figure 2.4: Multiplication of two sparse matrices

We would like to multiply two sparse matrices a and b represented as ordered lists as in Figure 2.3. We need to compute the elements of d by rows so that we can store them in their proper place without moving previously computed elements. To do this we pick a row of a and find all elements in column j of b for $j = 0, 1, \dots, b.cols - 1$, where $b.cols$ is the number of columns in b . Normally, we would have to scan all of b to find all the elements in column j . However, we can avoid this by first computing the transpose of b . This puts all column elements in consecutive order. Once we have located the elements of row i of A and column j of b , we just do a merge operation similar to that used for polynomial addition in Section 2.3. An alternative approach is explored in the exercises.

Before we write a matrix multiplication function, it will be useful to define the function of Program 2.12, which stores a matrix term. This function uses another function *ChangeSize 1D* (Program 2.13), which changes the size of a 1-dimensional array.

```
void SparseMatrix::StoreSum (const int sum, const int r, const int c)
// If sum != 0, then it along with its row and column position are stored as the
// last term in *this.
{
    if (sum != 0) {
        if (termu == capacity)
            ChangeSize 1D(2*capacity); // double size
        smArray[termu].row = r;
        smArray[termu].col = c;
        smArray[termu++].value = sum;
    }
}
```

Program 2.12: Storing a matrix term

The function *Multiply* (Program 2.14) multiplies the sparse matrices a (*this) and b to obtain the product matrix d using the strategy outlined above.

```

void ChangeSize(D,const int newSize)
{
    // Change the size of ansArray to newSize.
    if (newSize < anses) throw "New size must be >= number of terms";
    MatrixTerm *temp = new MatrixTerm[newSize]; // N new array
    copy(ansArray, ansArray + anses, temp);
    delete [] ansArray; // deallocate old memory
    ansArray = temp;
    capacity = newSize;
}

```

Program 2.13: Change the size of a 1-dimensional array

Function `Multiply` (which is invoked as `a.Multiply(b)`) makes use of variables `currRowIndex`, `currColIndex`, `currRowA`, `currColB`, and `currRowBBegin`. The variable `currRowA` is the row of *a* that is currently being multiplied with the columns of *b*; `currRowBBegin` is the position in *a* of the first element of row `currRowA`; and `currColB` is the column of *b* that is currently being multiplied with row `currRowA` of *a*. The variables `currRowIndex` and `currColIndex` are used to examine successive elements of row `currRowA` and column `currColB` of *a* and *b*, respectively. If matrices *a* and *b* are incompatible, then `Multiply` throws an exception in Line 3. Lines 10–14 of the function introduce a dummy term into each of *a* and *b*’s `anses`. This enables us to handle end conditions (i.e., computations involving the last row of *a* or last column of *b*) in an elegant way. Notice that because of the use of array doubling in function `StoreSum`, the capacity of `ansArray` may exceed the number of terms in the matrix. Also, because of the addition of the dummy term to `*this` its capacity will be more than the number of terms in `*this`. We can free the unused space in these two matrices by using the function `ChangeSize(1)`.

```

1 SparseMatrix SparseMatrix::Multiply(SparseMatrix b)
2 // Return the product of the sparse matrices *this and b.
3 if (cols != b.rows) throw "Incompatible matrices";
4 SparseMatrix bCopy = b.FastTranspose();
5 SparseMatrix d(rows, b.cols, 0);
6 int currRowIndex = 0;
7     currRowBegin = 0;
8     currRowA = ansArray[0].row;
9 // set boundary conditions
10 if (lines == capacity) ChangeSize(1); // anses + 1;

```

```

11  bXpose.ChangeSize(d(bXpose.terms + 1));
12  smArray[terms].row = rows;
13  bXpose.smArray[b.terms].row = b.col;
14  bXpose.smArray[b.terms].col = -1;
15  int sum = 0;
16  while (currRowIndex < terms)
17  { // generate row currRowA of d
18    int currColB = bXpose.smArray[0].row;
19    int currColIndex = 0;
20    while (currColIndex <= b.terms)
21    { // multiply row currRowA of *this by column currColB of b
22      if (smArray[currRowIndex].row != currRowA)
23        { // end of row currRowA
24          d.StoreSum(sum, currRowA, currColB);
25          sum = 0; // reset sum
26          currRowIndex = currRowBegin;
27          // advance to next column
28          while (bXpose.smArray[currColIndex].row == currColB)
29            currColIndex++;
30          currColB = bXpose.smArray[currColIndex].row;
31        }
32      else if (bXpose.smArray[currColIndex].row != currColB)
33        { // end of column currColB of b
34          d.StoreSum(sum, currRowA, currColB);
35          sum = 0; // reset sum
36          // set to multiply row currRowA with next column
37          currRowIndex = currRowBegin;
38          currColB = bXpose.smArray[currColIndex].row;
39        }
40      else
41        if (smArray[currRowIndex].col <
42            bXpose.smArray[currColIndex].col)
43          currRowIndex++; // advance to next term in row
44        else if (smArray[currRowIndex].col ==
45            bXpose.smArray[currColIndex].col)
46          { // add to sum
47            sum += smArray[currRowIndex].value +
48                bXpose.smArray[currColIndex].value;
49            currRowIndex++; currColIndex++;
50          }
51        else currColIndex++; // next term in currColB
52    } // end of while (currColIndex <= b.terms)

```

```

53   while (matArray[currRowIndex].row == currRowA) // advance to next row
54     currRowIndex++;
55   currRowBegin = currRowIndex;
56   currRowA = matArray[currRowIndex].row;
57   } // end of while (currRowIndex < numRows)
58   return a;
59 }

```

Program 2.14: Multiplying sparse matrices

Analysis of *Multiply*: We leave the correctness proof of this algorithm as an exercise. Let us examine its computing time. In addition to the space needed for a (≠this), b , d , and some simple variables, space is needed for the transpose matrix $bXpose$. Algorithm *FastTranspose* also needs some additional space. The exercises explore a strategy for *Multiply* that does not explicitly compute $bXpose$, and the only additional space needed is the same as that required by *FastTranspose*. Turning our attention to the computing time of *Multiply*, we see that lines 3-15 require only $O(b.cols + b.rows)$ time. The while loop of lines 16-57 is executed at most $a.rows$ times (once for each row of a). In each iteration of the while loop of lines 20-52, the value of *currRowIndex* or *currColB* or both increases by 1, or *currRowIndex* and *currColB* are reset. The maximum total increment in *currColB* over the whole loop is $b.rows$. If r_i is the number of terms in row r of a , then the value of *currRowIndex* can increase at most r_i times before *currRowIndex* moves to the next row of a . When this happens, *currRowIndex* is reset to *currRowBegin* in line 26. At the same time *currColB* is advanced to the next column. Hence, this resetting can take place at most $b.cols$ times. The total maximum increment in *currRowIndex* is therefore $b.cols \cdot r_i$. The maximum number of iterations of the while loop of lines 20-52 is therefore $b.cols + b.cols \cdot r_i + b.rows$. The time for this loop when multiplying with row r of a is $O(b.cols \cdot r_i + b.rows)$. Lines 53-54 take only $O(r_i)$ time. Hence, the time for the outer while loop, lines 16-57, for the iteration with row *currRowA* of a , is $O(b.cols \cdot r_i + b.rows)$. The overall time for this loop is then $O(\sum_i (b.cols \cdot r_i + b.rows)) = O(b.cols \cdot a.rows + a.rows \cdot b.rows)$.

Once again, we may compare the computing time with the time to multiply matrices when arrays are used. The classical multiplication algorithm is:

```

for (int i = 0; i < a.rows; i++)
    for (int j = 0; j < b.cols; j++)
    {
        sum = 0;
        for (int k = 0; k < a.cols; k++)
            sum += a[i][k] * b[k][j];
        c[i][j] = sum;
    }

```

The time for this is $O(a.rows \cdot a.cols \cdot b.cols)$. Since $a.terms \leq a.rows \cdot a.cols$ and $b.terms \leq a.cols \cdot b.rows$, the time for *Multiply* is at most $O(a.rows \cdot a.cols \cdot b.cols)$. However, its constant factor is greater than that for matrix multiplication using arrays. In the worst case, when $a.terms = a.rows \cdot a.cols$ or $b.terms = a.cols \cdot b.rows$, *Multiply* will be slower by a constant factor. However, when $a.terms$ and $b.terms$ are sufficiently smaller than their maximum values, i.e., a and b are sparse, *Multiply* will outperform the above multiplication algorithm for arrays.

The above analysis for *Multiply* is nontrivial. It introduces some new concepts in algorithm analysis and you should make sure you understand the analysis. \square

EXERCISES

1. How much time does it take to locate an arbitrary element $A[i][j]$ in the representation of this section and to change its value?
2. Analyze carefully the computing time and storage requirements of function *FastTranspose* (Program 2.11). What can you say about the existence of an even faster algorithm?
3. Write C++ functions to input and output a sparse matrix. These should be implemented by overloading the $>>$ and $<<$ operators. You should design the input and output formats. However, the internal representation should be a one dimensional array of nonzero terms as used in this section. Analyze the computing time of your functions.
4. Rewrite function *FastTranspose* (Program 2.11) so that it uses only one array rather than the two arrays required to hold *RowSize* and *RowStart*.
5. Develop a correctness proof for function *Multiply* (Program 2.14).
6. Use the row pointers idea used in *FastTranspose* and rewrite *Multiply* (Program 2.14) to multiply two sparse matrices represented as in Section 2.4 without explicitly transposing either. What is the computing time of your algorithm?

7. A variation of the scheme discussed in this section for sparse matrix representation involves representing only the nonzero terms in a one-dimensional array v in the order described. In addition, a strip of $n \times n$ bits, $\text{bits}[n][n]$ is also kept. $\text{bits}[i][j] = 0$ if $a[i][j] = 0$, and $\text{bits}[i][j] = 1$ if $a[i][j] \neq 0$. The figure below illustrates the representation for the sparse matrix of Figure 2.2(b).

$\begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 15 \\ 22 \\ -15 \\ 11 \\ 3 \\ -6 \\ 91 \\ 28 \end{bmatrix}$
--	--

- On a computer with w bits per word, how much storage is needed to represent an $n \times n$ sparse matrix that has t nonzero terms?
- Write an algorithm to add two sparse matrices represented as above. How much time does your algorithm take?
- Discuss the merits of this representation versus the representation of Section 2.4. Consider space and time requirements for such operations as random access, add, multiply, and transpose. Note that the random access time can be improved somewhat by keeping another array rs such that $rs[i] = \text{number of nonzero terms in rows } i \text{ through } i-1$.

2.5 REPRESENTATION OF ARRAYS

Multidimensional arrays are usually implemented by storing the elements in a one-dimensional array. In this section, we develop a representation in which an arbitrary array element, say $a[i_1][i_2] \dots [i_n]$, gets mapped onto a position in a one-dimensional C++ array so that it can be retrieved efficiently. This is necessary since programs using arrays may, in general, use array elements in a random order. In addition to being able to retrieve array elements easily, it is also necessary to be able to determine the amount of memory to be reserved for a particular array. Assuming that each array element requires only one word of memory, the number of words needed is the number of elements in the array. If an array is declared $a[a_1][a_2] \dots [a_n]$, where 0 through a_i-1 is the range of index values in dimension i , then it is easy to see that the number of elements is

$$\prod_{i=0}^n u_i$$

One of the common ways to represent an array is in *row major order* (see Exercise 3 for *column major order*). If we have the declaration

`a[2][3][2][2]`

then we have a total of $2 \times 3 \times 2 \times 2 = 24$ elements. Using row major order, these elements will be stored as

`a[0][0][0][0], a[0][0][0][1], a[0][0][1][0], a[0][0][1][1]`

and continuing

`a[0][1][0][0], a[0][1][0][1], a[0][1][1][0], a[0][1][1][1]`

for three more sets of four until we get

`a[1][2][0][0], a[1][2][0][1], a[1][2][1][0], a[1][2][1][1]`

We see that the index at the right moves the fastest. In fact, if we view the indices as numbers, we see that they are, in some sense, increasing:

0000, 0001, ..., 1210, 1211

A synonym for row major order is *lexicographic order*.

The problem is how to translate from the name `a[i1][i2] ... [in]` to the correct location in the one-dimensional array. Suppose `a[0][0][0][0]` is stored at position 0. Then `a[0][0][0][1]` will be at position 1 and `a[1][2][1][1]` at position 23. These two addresses are easy to guess. In general, we can derive a formula for the address of any element. This formula makes use of only the starting address of the array plus the declared dimensions.

Before obtaining a formula for the case of an n -dimensional array, let us look at the row major representation of one-, two-, and three-dimensional arrays. To begin with, if a is declared `a[u1]`, then assuming one word per element, it may be represented in sequential memory as in Figure 2.5. If α is the address of `a[0]`, then the address of an arbitrary element `a[i]` is just $\alpha + i$.

The two-dimensional array `a[u1][u2]` may be interpreted as u_1 rows, `row0, row1, ..., rowu1-1`, each row consisting of u_2 elements. In a row major representation, these rows would be represented in memory as in Figure 2.6.

Again, if α is the address of `a[0][0]`, then the address of `a[i][0]` is

array element:	$a[0]$	$a[1]$	$a[2]$	\dots	$a[i]$	\dots	$a[u_1-1]$
address:	α	$\alpha+1$	$\alpha+2$	\dots	$\alpha+i$	\dots	$\alpha+u_1-1$

Figure 2.5: Sequential representation of $a[u_1]$

	col 0	col 1	\dots	col u_2-1
row 0	X	X	\dots	X
row 1	X	X	\dots	X
row 2	X	X	\dots	X
\vdots	\vdots	\vdots	\vdots	\vdots
row u_1-1	X	X	\dots	X

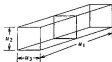
(a)

Figure 2.6: Sequential representation of $a[u_1][u_2]$

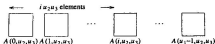
$\alpha + i * u_2$, as there are i rows, each of size u_2 , preceding the first element in the i th row. Knowing the address of $a[i][0]$, we can say that the address of $a[i][j]$ is then simply $\alpha + i * u_2 + j$.

Figure 2.7 shows the representation of the three-dimensional array $a[u_1][u_2][u_3]$. This array is interpreted as a 1 two-dimensional arrays of dimension $u_2 \times u_3$. To locate $a[i][j][k]$, we first obtain $\alpha + i * u_2 * u_3$ as the address for $a[i][0][0]$ since there are i two-dimensional arrays of size $u_2 \times u_3$ preceding this element. From this and the formula for addressing a two-dimensional array, we obtain $\alpha + i * u_2 * u_3 + j * u_3 + k$ as the address of $a[i][j][k]$.

Generalizing on the preceding discussion, the addressing formula for any



(a) 3-dimensional array $A[u_1][u_2][u_3]$ regarded as u_1 2-dimensional arrays



(b) Sequential row major representation of a 3-dimensional array. Each 2-dimensional array is represented as in Figure 2.6

Figure 2.7: Sequential representation of a $a[u_1][u_2][u_3]$

element $a[i_1][i_2] \cdots [i_n]$ in an n -dimensional array declared as $a[u_1][u_2] \cdots [u_n]$ may be easily obtained. If α is the address for $a[0][0] \cdots [0]$ then $\alpha + i_1 u_2 u_3 \cdots u_n$ is the address for $a[i_1][0] \cdots [0]$. The address for $a[i_1][i_2][0] \cdots [0]$ is then $\alpha + i_1 u_2 u_3 \cdots u_n + i_2 u_3 u_4 \cdots u_n$.

Repeating in this way, the address for $a[i_1][i_2] \cdots [i_n]$ is

$$\begin{aligned}
 & a + i_1 u_1 u_2 \cdots u_n \\
 & + i_2 u_2 u_3 \cdots u_n \\
 & + i_3 u_3 u_4 \cdots u_n \\
 & \vdots \\
 & + i_{n-1} u_n \\
 & + i_n
 \end{aligned}$$

$$= a + \sum_{j=1}^n i_j u_j \text{ where } \begin{cases} a_j = \prod_{k=j+1}^n u_k & 1 \leq j < n \\ a_n = 1 \end{cases}$$

Note that a_j may be computed from a_{j+1} , $1 \leq j < n$, using only one multiplication as $a_j = u_{j+1} a_{j+1}$. Thus, a compiler will initially take the declared bounds u_1, \dots, u_n and use them to compute the constants a_1, \dots, a_{n-1} using $n-2$ multiplications. The address of $a[i_1], \dots, [i_n]$ can then be found using the formula, requiring $n-1$ more multiplications and n additions and n subtractions.

EXERCISES

1. [Programming Project] Even though the multidimensional array is provided as a standard data object in C++, it is often useful to define your own class for multidimensional arrays. This gives a more robust class that:
 - (a) Performs range checking.
 - (b) Does not require the index set in each dimension to consist of consecutive integers starting at 0.
 - (c) Allows array assignment.
 - (d) Allows initialization of an array with another array.
 - (e) Selects the range in each dimension of the array during runtime.
 - (f) Allows dynamic modification of the range and size of the array.
 - (g) Provides a mechanism to determine the size of an array.

Implement a class `stdArray` that stores floating point elements and provides the functionality specified above. You will need to define two pointer data members corresponding to one-dimensional arrays, which will be dynamically created: the array of elements, and an array that stores the upper and lower bounds of each dimension. Additional data members may be

required. The data members of your class must be initialized by a constructor that takes as input the number of dimensions and the range of each dimension. Also, provide operations to read/write array elements from/to a file.

2. How many values can be held by each of the arrays $a[n]$, $b[n][n]$, $c[n][2][3]$?
3. Obtain an addressing formula for the element $a[i_1][i_2], \dots, [i_n]$ in an array declared as $a[u_1][u_2], \dots, [u_n]$. Assume a column-major representation of the array with one word per element and α the address of $a[0][0], \dots, [0]$. In a column-major representation, a two-dimensional array is stored sequentially by columns rather than by rows.

2.6 THE STRING ABSTRACT DATA TYPE

In this section, we turn our attention to a data type, the string, whose component elements are characters. As an ADT, we define a string to have the form $S = x_0, \dots, x_{n-1}$, where x_i are characters taken from the character set of the programming language, and n is the length of the string. If $n = 0$, then S is an empty or null string.

There are several useful operations we could specify for strings. Some of these operations are similar to those required for other ADTs: creating a new empty string, reading a string or printing it out, appending two strings together (called *concatenation*), or copying a string. However, there are other operations that are unique to our new ADT, including comparing strings, inserting a substring into a string, removing a substring from a string, or finding a pattern in a string. We have listed some of the essential operations in ADT 2.5, which contains our specification of the string ADT.

C++ includes a *string* class that provides many more functions than are specified in our ADT. We do not go into the details of this C++ class here. Instead, we focus on the problem of string pattern matching (i.e., the function *Find* specified in ADT 2.5). In the remainder of this section, we assume that, in our string class, strings are represented by the private data member *str* of type *char**. We may access the *i*th character of *str* using either of the notations *str[i]* and **(str + i)* and we may assign a string to *str* using a statement such as *str = "abc"*.

```

class String
{
public:
    String(char *init, int n);
    // Constructor that initializes *this to string (init of length n.

    bool operator==(String t);
    // If the string represented by *this equals t, return true;
    // else return false.

    bool operator!();
    // If *this is empty then return true; else return false.

    int Length();
    // Return the number of characters in *this.

    String Concat(String t);
    // Return a string whose elements are those of *this followed by those of t.

    String Substr(int i, int j);
    // Return a string containing the j characters of *this at positions i, i+1, ...,
    // i + j - 1 if these are valid positions of *this; otherwise, throw an exception.

    int Find(String pat);
    // Return an index i such that pat matches the substring of *this that begins
    // at position i. Return -1 if pat is either empty or not a substring of *this.
};

```

ADT 2.5: Abstract data type String

2.6.1 String Pattern Matching: A Simple Algorithm

Assume that we have two strings, s and pat , where pat is a pattern to be searched for in s . We will determine if pat is in s by using the function *Find*. The invocation $s.Find(pat)$ returns an index i such that pat matches the substring of s that begins at position i . It returns -1 if and only if pat is either empty or is not a substring of s . Let us examine how a function *Find* may be implemented.

The easiest but least efficient method to determine whether pat is in s is to serially consider each position of s and determine if this position is the starting point of a match. Let $lengthP$ and $lengthS$ denote the lengths of the pattern pat and the string s , respectively. Positions of s to the right of position $lengthS - lengthP$ need not be considered, as there are not enough characters to their right to complete a match with pat . Function *Find* (Program 2.15)

implements this strategy. The complexity of this function is $O(\text{length}^P \cdot \text{length}^S)$.

```
int String::Find(String pat)
// Return -1 if pat does not occur in *this;
// otherwise return the first position in *this, where pat begins.
    for (int start = 0; start <= Length() - pat.Length(); start++)
        // check for match beginning at str[start]
        {
            int j;
            for (j = 0; j < pat.Length() && str[start + j] == pat.str[j]; j++)
                if (j == pat.Length()) return start; // match found
            // no match at position start
        }
    return -1; // pat is empty or does not occur in s
}
```

Program 2.15: Exhaustive pattern matching

We can introduce heuristics into function *Find* that improve its performance on certain pairs of *s* and *pat*. For example, for each position *start* of *s* considered in function *Find*, we may check for a match of the last character of *pat* with the character at position *start* + *length*^{*P*} - 1 of *s* before examining characters 0 through *length*^{*P*} - 2 of *pat* for a match. The asymptotic complexity of the resulting pattern matching function is still $O(\text{length}^P \cdot \text{length}^S)$.

2.6.2 String Pattern Matching: The Knuth-Morris-Pratt Algorithm

Ideally, we would like an algorithm that works in $O(\text{length}^P + \text{length}^S)$ time. This is optimal for this problem, as in the worst case it is necessary to look at all characters in the pattern and string at least once. We want to search the string for the pattern without moving backwards in the string. That is, if a mismatch occurs we want to use our knowledge of the characters in the pattern and the position in the pattern, where the mismatch occurred to determine where we should continue the search. Knuth, Morris, and Pratt have developed a pattern-matching algorithm that works in this way and has linear complexity. Using their example, suppose

$$\text{pat} = a b c a b c u c a b$$

Let $s = s_0 s_1 \cdots s_{n-1}$ be the string and assume that we are currently

determining whether or not there is a match beginning at s_i . If $s_i \neq a$, then clearly we may proceed by comparing s_{i+1} and a . Similarly, if $s_i = a$ and $s_{i+1} \neq b$, then we may proceed by comparing s_{i+1} and a . If $s_i s_{i+1} = ab$ and $s_{i+2} \neq c$ then we have the situation

$s =$	-	a	b	?	?	?	?
$pat =$		a	b	c	a	b	c	a	c	a	b

The ? implies that we do not know what the character in s is. The first ? in s represents s_{i+2} , where $s_{i+2} \neq c$. At this point we know that we may continue the search for a match by comparing the first character in pat with s_{i+2} . There is no need to compare this character of pat with s_{i+1} , since we already know that s_{i+1} is the same as the second character of pat , b , and so $s_{i+1} \neq a$. Let us try this again assuming a match of the first four characters in pat followed by a mismatch, i.e., $s_{i+4} \neq b$. We now have the situation

$s =$	-	a	b	c	a	?	?	.	.	.	?
$pat =$		a	b	c	a	b	c	a	c	a	b

We observe that the search for a match can proceed by comparing s_{i+4} and the second character in pat , b . This is the first place a partial match can occur by sliding the pattern pat towards the right. Thus, by knowing the characters in the pattern and the position in the pattern where a mismatch occurs with a character in s , we can determine where in the pattern to continue the search for a match without moving backwards in s . To formalize this, we define a failure function for a pattern.

Definition: If $p = p_0 p_1 \cdots p_{n-1}$ is a pattern, then its failure function, f , is defined as

$$f(j) = \begin{cases} \text{largest } k < j \text{ such that } p_0 \cdots p_k = p_{j-k} \cdots p_j & \text{if such a } k \geq 0 \text{ exists} \\ -1 & \text{otherwise.} \end{cases} \quad \square$$

For the example pattern above, $pat = abcabcacab$, we have

j	0	1	2	3	4	5	6	7	8	9
pat	a	b	c	a	b	c	a	c	a	b
f	-1	-1	-1	0	1	2	3	-1	0	1

From the definition of the failure function, we arrive at the following rule for pattern matching: If a partial match is found such that $s_{i-j} \cdots s_{i-1} = p_0 \cdots p_{j-1}$ and $s_i \neq p_j$ then matching may be resumed by

comparing s_j and $p_{j(j-1)+1}$ if $j \neq 0$. If $j = 0$, then we may continue by comparing s_{j+1} and p_j . This pattern-matching rule translates to function *FastFind* (Program 2.16). *FastFind* uses an array of ints, f , to represent the failure function. f is a private data member of *String*.

```

1 int String::FastFind(String par)
2 // Determine if par is a substring of s.
3 int posP = 0, posS = 0;
4 int lengthP = par.Length(), lengthS = Length();
5 while ((posP < lengthP) && (posS < lengthS))
6     if (par.str[posP] == str[posS]) { // character match
7         posP++; posS++;
8     }
9     else
10        if (posP == 0)
11            posS++;
12        else posP = par.f[posP - 1] + 1;
13 if (posP < lengthP) return -1;
14 else return posS - lengthP;
15 }

```

Program 2.16: Pattern-matching with a failure function

Analysis of *FastFind*: The correctness of *FastFind* follows from the definition of the failure function. To determine the computing time, we observe that lines 7 and 11 can be executed for a total of at most lengthS times, since in each iteration posS is incremented by 1 but posS is never decremented in the algorithm. As a result, posP can move right on par at most lengthS times (line 7). Since each execution of line 12 moves posP left on par , it follows that this clause can be executed at most lengthS times, since otherwise posP becomes less than 0. Consequently, the maximum number of iterations of the while loop is lengthS , and the computing time of *FastFind* is $O(\text{lengthS})$. \square

From the analysis of *FastFind*, it follows that if we can compute the failure function in $O(\text{lengthP})$ time, then the entire pattern-matching process will have a computing time proportional to the sum of the lengths of the string and pattern. Fortunately, there is a fast way to compute the failure function. This is based upon the following restatement of the failure function:

$$f(j) = \begin{cases} -1 & \text{if } j = 0 \\ j^m(j-1) + 1 & \text{where } m \text{ is the least integer } k \text{ for which } p_{j^k(j-1)+1} = p_j \\ -1 & \text{if there is no } k \text{ satisfying the above} \end{cases}$$

(Note that $f^k(j) = f(f^{k-1}(j))$ and $f^0(j) = f(f^{-1}(j))$). This directly yields the function of Program 2.17 to compute f .

```

1 void String::FailureFunction()
2 { // Compute the failure function for the pattern *this.
3   int lengthP = Length();
4   f[0] = -1;
5   for (int j = 1; j < lengthP; j++) // compute f[j]
6   {
7     int i = f[j-1];
8     while ((*str + j) != *(str+i+1) && (i >= 0)) i = f[i];
9     if (*str + j == *(str+i+1))
10      f[j] = i + 1;
11    else f[j] = -1;
12  }
13 }

```

Program 2.17: Computing the failure function

Analysis of FailureFunction: In each iteration of the **while** loop the value of i decreases (by the definition of f). The variable i is reset at the beginning of each iteration of the **for** loop. However, it is either reset to -1 (when $j = 1$ or when the previous iteration went through line 11), or it is reset to a value i greater than its terminal value on the previous iteration (i.e., when the previous iteration went through line 10). Since only $\text{lengthP} - 1$ executions of line 7 are made, the value of i therefore has a total increment of at most $\text{lengthP} - 1$. Hence it cannot be decremented more than $\text{lengthP} - 1$ times. Consequently, the **while** loop is iterated at most $\text{lengthP} - 1$ times over the whole algorithm, and the computing time of *FailureFunction* is $O(\text{lengthP})$. \square

Note that when the failure function is not known in advance, pattern matching can be carried out in time $O(\text{lengthP} + \text{lengthS})$ by first computing *FailureFunction* and then performing a pattern match using function *FastFind*.

EXERCISES

1. Write a function *String::Frequency* that determines the frequency of occurrence of each of the distinct characters in the string. Test your function using suitable data.

- Write a function, *String::Delete*, that accepts two integers, *start* and *length*. The function computes a new string that is equivalent to the original string, except that *length* characters beginning at *start* have been removed.
- Write a function, *String::CharDelete*, that accepts a character *c*. The function returns the string with all occurrences of *c* removed.
- Write a function to make an in-place replacement of a substring *w* of a string by the string *x*. Note that *w* may not be of the same size as *x*. What is the complexity of your function?
- If $x = (x_0, \dots, x_{n-1})$ and $y = (y_0, \dots, y_{n-1})$ are strings, where x_i and y_i are letters of the alphabet, then x is less than y if $x_i = y_i$ for $0 \leq i < j$ and $x_j < y_j$ or if $x_i = y_i$ for $0 \leq i \leq m$ and $m < n$. Write an algorithm that takes two strings x, y and returns either -1 , 0 , or $+1$ if $x < y$, $x = y$, or $x > y$ respectively.
- Find a string and a pattern for which function *Find* (Program 2.15) takes time proportional to $\text{length}P \cdot \text{length}S$.
 - Do part (a) under the assumption that the function *Find* has been modified to check for a match with the last character of the pattern first (see text for an explanation of this heuristic).
- Compute the failure function for each of the following patterns:
 - $a a a a b$
 - $a b a b a a$
 - $a b a a b a a b b$
- Let $p_0 p_1 \dots p_{n-1}$ be a pattern of length n . Let f be its failure function. Define $f^1(j) = f(j)$ and $f^m(j) = f(f^{m-1}(j))$, $0 \leq j < n$ and $m > 1$. Show, using the definition of f , that

$$f(j) = \begin{cases} -1 & \text{if } j = 0 \\ f^m(j-1)+1 & \text{where } m \text{ is the least integer } k \text{ for which } p_{f^{k-1}(j)+1} = p_j \\ -1 & \text{if there is no } k \text{ satisfying the above} \end{cases}$$

- The definition of the failure function may be strengthened to

$$f(j) = \begin{cases} \text{largest } i < j \text{ such that } p_0 \dots p_i = p_{j-i} \dots p_j \text{ and } p_{i+1} \neq p_{j+1} \\ -1 & \text{if there is no } i \geq 0 \text{ satisfying above} \end{cases}$$

- Obtain the new failure function for the pattern *par* of the text.
- Show that if this definition for f is used, then algorithm *FastFind*

(Program 2.16) still works correctly.

- (c) Modify algorithm *FailureFunction* (Program 2.17) to compute f under this definition. Show that the computing time is still $O(n)$.
- (d) Are there any patterns for which the observed computing time of *FastFind* is more with the new definition of f than with the old one? Are there any for which it is less? Give examples.

2.7 REFERENCES AND SELECTED READINGS

The Knuth, Morris, Pratt pattern-matching algorithm can be found in "Fast pattern matching in strings," *SIAM Journal on Computing*, 6:2, 1977, pp. 323-350. A discussion of the Knuth Morris Pratt algorithm, along with other string matching algorithms, may be found in *Introduction to Algorithms* Second Edition, by T. Cormen, C. Leiserson, R. Rivest and C. Stein, McGraw Hill, New York, 2002.

2.8 ADDITIONAL EXERCISES

- Write a C++ function to make an in-place reversal of the order of elements in the array *list*. That is, the function should transform *list* such that following the transformation, *list*[i] contains the element originally in *list*[$n - i - 1$], where n is the total number of elements in *list*. The only additional space available to your function is that for simple variables. How much time does your function take to accomplish the reversal?
- An $n \times n$ matrix is said to have a saddle point if some entry $a[i][j]$ is the smallest value in row i and the largest value in column j . Write a C++ function that determines the location of a saddle point if one exists. What is the computing time of your method?
- When all the elements either above or below the main diagonal of a square matrix are zero, then the matrix is said to be triangular. Figure 2.8 shows a lower and an upper triangular matrix. In a lower triangular matrix, a , with n rows, the maximum number of nonzero terms in row i is $i + 1$. Hence, the total number of nonzero terms is $\sum_{i=0}^{n-1} (i + 1) = n(n + 1)/2$. For large n it would be worthwhile to save the space taken by the zero entries in the upper triangle. Obtain an addressing formula for elements a_{ij} in the lower triangle if this lower triangle is stored by rows in an array $b[n(n + 1)/2]$ with $a[0][0]$ being stored in $b[0]$. What is the relationship between i and j for elements in the zero part of A ?

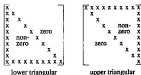


Figure 2.8: Lower and upper triangular matrices

- Let a and b be two lower triangular matrices, each with n rows. The total number of elements in the lower triangles is $n(n+1)$. Devise a scheme to represent both the triangles in an array $c[n][n+1]$. [Hint: Represent the triangle of a as the lower triangle of c and the transpose of b as the upper triangle of c .] Write algorithms to determine the values of $a[i][j]$ and $b[j][i]$, $0 \leq i, j < n$, from the array c .
- Another kind of sparse matrix that arises often in practice is the tridiagonal matrix. In this square matrix, all elements other than those on the major diagonal and on the diagonals immediately above and below this one are zero (Figure 2.9).



Figure 2.9: Tridiagonal matrix

If the elements in the band formed by these three diagonals are represented

122 Arrays

by rows in an array, b , with $a[0][0]$ being stored in $b[0]$, obtain an algorithm to determine the value of $a[i][j]$, $0 \leq i, j < n$ from the array b .

6. A square band matrix $a_{n,n}$ is an $n \times n$ matrix in which all the nonzero terms lie in a band centered around the main diagonal. The band includes $a-1$ diagonals below and above the main diagonal and also the main diagonal (Figure 2.10).

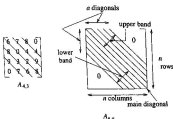


Figure 2.10: Square band matrix.

- How many elements are there in the band of $a_{n,n}$?
- What is the relationship between i and j for elements a_{ij} in the band of $a_{n,n}$?
- Assume that the band of $a_{n,n}$ is stored sequentially in an array b by diagonals starting with the lowermost diagonal. Thus, $a_{4,5}$ above would have the following representation:

$b[0]$	$b[1]$	$b[2]$	$b[3]$	$b[4]$	$b[5]$	$b[6]$	$b[7]$	$b[8]$	$b[9]$	$b[10]$	$b[11]$	$b[12]$	$b[13]$
9	7	8	3	6	4	0	2	8	7	4	9	8	4
a_{20}	a_{31}	a_{10}	a_{21}	a_{32}	a_{00}	a_{11}	a_{22}	a_{33}	a_{01}	a_{12}	a_{23}	a_{02}	a_{13}

Obtain an addressing formula for the location of an element a_{ij} in the lower band of $a_{n,n}$, e.g., $LOC(a_{20}) = 0$, $LOC(a_{33}) = 1$ in the example above.

7. A generalized band matrix $a_{n,a,b}$ is an $n \times n$ matrix a in which all the nonzero terms lie in a band made up of $a - 1$ diagonals below the main diagonal, the main diagonal, and $b - 1$ diagonals above the main diagonal (Figure 2.11).
- How many elements are there in the band of $a_{n,a,b}$?
 - What is the relationship between i and j for elements a_{ij} in the band of $a_{n,a,b}$?
 - Obtain a sequential representation of the band of $a_{n,a,b}$ in the one-dimensional array c . For this representation, write a C++ function value (n,a,b,i,j,c) that determines the value of element a_{ij} in the matrix $a_{n,a,b}$. The band of $a_{n,a,b}$ is represented in the array c .

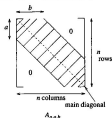


Figure 2.11: Generalized band matrix

8. [Programming Project] There are a number of problems, known collectively as "random walk" problems, that have been of longstanding interest to the mathematical community. All but the most simple of these are extremely difficult to solve and for the most part they remain largely unsolved. One such problem may be stated as:

A (drunken) cockroach is placed on a given square in the middle of a

tile floor in a rectangular room of size $n \times m$ tiles. The bug wanders (possibly in search of an aspirin) randomly from tile to tile throughout the room. Assuming that he may move from his present tile to any of the eight tiles surrounding him (unless he is against a wall) with equal probability, how long will it take him to touch every tile on the floor at least once?

Hard as this problem may be to solve by probability theory techniques, it is easy to solve using the computer. The technique for doing so is called "simulation." This technique is widely used in industry to predict traffic flow, inventory control, and so forth. The problem may be simulated using the following method:

An $n \times m$ array *count* is used to represent the number of times our cockroach has reached each tile on the floor. All the cells of this array are initialized to zero. The position of the bug on the floor is represented by the coordinates (*i*bug, *j*bug). The eight possible moves of the bug are represented by the tiles located at (*i*bug + *imove*[*k*], *j*bug + *jmove*[*k*]), where $0 \leq k \leq 7$ and

<i>imove</i> [0] =	-1	<i>jmove</i> [0] =	1
<i>imove</i> [1] =	0	<i>jmove</i> [1] =	1
<i>imove</i> [2] =	1	<i>jmove</i> [2] =	1
<i>imove</i> [3] =	1	<i>jmove</i> [3] =	0
<i>imove</i> [4] =	1	<i>jmove</i> [4] =	-1
<i>imove</i> [5] =	0	<i>jmove</i> [5] =	-1
<i>imove</i> [6] =	-1	<i>jmove</i> [6] =	-1
<i>imove</i> [7] =	-1	<i>jmove</i> [7] =	0

A random walk to one of the eight given squares is simulated by generating a random value for *k* lying between 0 and 7. Of course, the bug cannot move outside the room, so coordinates that lead up a wall must be ignored and a new random combination formed. Each time a square is entered, the count for that square is incremented so that a nonzero entry shows the number of times the bug has landed on that square so far. When every square has been entered at least once, the experiment is complete.

Write a program to perform the specified simulation experiment. Your program MUST:

- handle all values of *n* and *m*, $2 \leq n \leq 40$, $2 \leq m \leq 20$
- perform the experiment for (1) $n = 15$, $m = 15$, starting point: (9,9) and (2) $n = 39$, $m = 19$, starting point: (0,0)

- (c) have an iteration limit, that is, a maximum number of squares the bug may enter during the experiment (this avoids getting hung in an infinite loop; a maximum of 50,000 is appropriate for this exercise)
- (d) for each experiment, print (1) the total number of legal moves that the cockroach makes and (2) the final *count* array (this will show the density of the walk, that is, the number of times each tile on the floor was touched during the experiment).

(Have an aspirin.) This exercise was contributed by Steve Olsen.

9. [Programming Project] Chess provides the setting for many fascinating diversions that are quite independent of the game itself. Many of these are based on the strange L-shaped move of the knight. A classical example is the problem of the knight's tour, which has captured the attention of mathematicians and puzzle enthusiasts since the beginning of the eighteenth century. Briefly stated, the problem is to move the knight, beginning from any given square on the chessboard, in such a manner that it travels successively to all 64 squares, touching each square once and only once. It is convenient to represent a solution by placing the numbers 1, 2, ..., 64 in the squares of the chessboard indicating the order in which the squares are reached. Note that it is not required that the knight be able to reach the initial position by one more move; if this is possible, the knight's tour is called *re-entrant*. One of the more ingenious methods for solving the problem of the knight's tour was that given by J. C. Warnsdorff in 1823. His rule was that the knight must always be moved to one of the squares from which there are the fewest exits to squares not already traversed. Use Warnsdorff's rule to construct a particular solution to the problem by hand before reading any further.

The most important decisions to be made in solving a problem of this type are those concerning how the data is to be represented in the computer. Perhaps the most natural way to represent the chessboard is by an 8×8 array *board*, as shown in Figure 2.12. The eight possible moves of a knight on square (4,2) are also shown in this figure. In general a knight at (i,j) may move to one of the squares $(i-2,j+1)$, $(i-1,j+2)$, $(i+1,j+2)$, $(i+2,j+1)$, $(i+2,j-1)$, $(i+1,j-2)$, $(i-1,j-2)$, $(i-2,j-1)$. Notice, however, that if (i,j) is located near one of the edges of the board, some of these possibilities could move the knight off the board, and, of course, this is not permitted. The eight possible knight moves may conveniently be represented by two arrays *knowx* and *knowy*:

	0	1	2	3	4	5	6	7
0								
1								
2		8		1				
3	7				2			
4			K					
5	6				3			
6		5		4				
7								

Figure 2.12: Legal moves for a knight

$k\text{mov}1$	$k\text{mov}2$
-2	1
-1	2
1	2
2	1
2	-1
1	-2
-1	-2
-2	-1

Then a knight at (i, j) may move to $(i + k\text{mov}1[k], j + k\text{mov}2[k])$, where k is some value between 0 and 7, provided that the new square lies on the chessboard.

An algorithm to solve the knight's tour problem using Warnsdorff's rule is described below.

- [Initialize chessboard] For $0 \leq i, j \leq 7$, set $\text{board}[i][j]$ to 0.
- [Set starting position] Read and print i, j and then set $\text{board}[i][j]$ to

- 1.
- (c) [Loop] For $2 \leq m \leq 64$, do steps (d) through (g).
- (d) [Form set of possible next squares] Test each of the eight squares one knight's move away from (i, j) and form a list of the possibilities for the next square ($next[i], next[j]$). Let $npot$ be the number of possibilities. (That is, after performing this step we will have $next[i] = i + kmove[k]$ and $next[j] = j + kmove2[k]$, for certain values of k between 0 and 7. Some of the squares $(i + kmove[k], j + kmove2[k])$ may be impossible for the next move either because they lie off the chessboard or because they have been previously occupied by the knight (i.e., they contain a nonzero number). In every case we will have $0 \leq npot \leq 8$.)
- (e) [Test special cases] If $npot = 0$, the knight's tour has come to a premature end; report failure and then go to step (h). If $npot = 1$, there is only one possibility for the next move; set $min = 0$ and go right to step (g).
- (f) [Find next square with minimum number of exits] For $1 \leq i \leq npot$ set $exits[i]$ to the number of exits from square $(next[i], next[j])$. That is, for each of the values of i , examine each of the next squares $(next[i] + kmove[k], next[j] + kmove2[k])$ to see if it is an exit from $(next[i], next[j])$, and count the number of such exits in $exits[i]$. (Recall that a square is an exit if it lies on the chessboard and has not been previously occupied by the knight.) Finally, set min to the location of the minimum value of $exits$. (There may be more than one occurrence of the minimum value of $exits$. If this happens, it is convenient to let min denote the first such occurrence, although it is important to realize that by so doing we are not actually guaranteed finding a solution. Nevertheless, the chances of finding a complete knight's tour in this way are remarkably good, and that is sufficient for the purposes of this exercise.)
- (g) [Move knight] Set $i = next[min]$, $j = next[j][min]$, and $board[i][j] = m$. Thus, (i, j) denotes the new position of the knight, and $board[i][j]$ records the move in proper sequence.
- (h) [Output] Output $board$ showing the solution to the knight's tour and then terminate the algorithm.

The problem is to write a C++ program that corresponds to this algorithm. This exercise was contributed by Legenhäuser and Rehrman.

CHAPTER 3

Stacks and Queues

3.1 Templates in C++

In this section, we introduce the concept of templates. This is a mechanism provided by C++ to make classes and functions more reusable. We shall discuss both template functions and template classes.

3.1.1 Template Functions

Consider function *SelectionSort* of Program 1.6, which sorts an array of integers using the selection sort method. Suppose, we now wish to use selection sort to sort an array of floating point numbers. The code of Program 1.6 can be used to do this if we change `int *a` to `float *a` in the function header. From a practical standpoint, this can be achieved by using a text editor to replicate the code for function *SelectionSort* and then making the stated change. This process would have to be repeated if we next wished to sort an array of, say, characters.

Considerable software development time and money can be saved if we

use templates or parameterized types. A template may be viewed as a variable that can be instantiated to any data type, irrespective of whether this data type is a fundamental C++ type or a user-defined type. The template function *SelectionSort* (Program 3.1) is defined using the parameterized type *T*.

```

1 template <class T>
2 void SelectionSort (T *a, const int n)
3 // Sort a[0] to a[n-1] into nondecreasing order.
4     for ( int i = 0; i < n ; i++ )
5     {
6         int j = i;
7         // find smallest integer in a[i] to a[n-1]
8         for ( int k = i + 1 ; k < n ; k++ )
9             if (a[k] < a[j]) j = k ;
10        swap (a[i], a[j]);
11    }
12 }
```

Program 3.1: Selection sort using templates

Function *SelectionSort* can now be used quite easily to sort an array of *ints* or *floats* as shown in Program 3.2. Function *SelectionSort* is instantiated to the type of the array argument that is supplied to it. For example, the first call to *SelectionSort* knows that it is to sort an array of floating point numbers because array *farray* is an array of floating point numbers.

```

float farray[100];
int intarray[250];

.

// assume that the arrays are initialized at this point
SelectionSort(farray, 100);
SelectionSort(intarray, 250);
```

Program 3.2: Code fragment illustrating template instantiation

Observe that *SelectionSort* uses the operator "<" to compare two objects of type *T* in line 9. So long as this operator (as well as others used by *swap*) is defined for the data type *T*, Program 3.1 may be used to sort an array of type *T*.

130 Stacks and Queues

Suppose we want to use Program 3.1 to sort an array of *Rectangle*s in non-decreasing order of their areas. We cannot instantiate *T* to *Rectangle* as we did with *int* and *float* in Program 3.2 because operator "<" of line 9 is undefined for *Rectangle*. This can be remedied by overloading operator< so that it returns *true* if the first operand has less area than the second and *false*, otherwise. Of course, additional operators used by *swap* also must be defined for *Rectangle* unless the C++ default (if any) for these operators will suffice.

A template function that we shall find quite useful in this text is one to change the size of a 1-dimensional array. We have already seen two applications for array resizing (polynomial and matrix addition (Programs 2.6 and 2.13)). Program 3.3 gives a template function that changes the size of a 1-dimensional array of type *T* from *oldSize* to *newSize*.

```
template <class T>
void ChangeSize1D(T*& a, const int oldSize, const int newSize)
{
    if (newSize < 0) throw "New length must be >= 0";

    T* temp = new T[newSize];           // new array
    int number = min(oldSize, newSize); // number to copy
    copy(a, a + number, temp);
    delete [] a;                        // deallocate old memory
    a = temp;
}
```

Program 3.3: Template function to change the size of a 1-dimensional array

3.1.2 Using Templates to Represent Container Classes

A *container class* is a class that represents a data structure that contains or stores a number of data objects. Objects can usually be added to or deleted from a container class. The array is an example of a container class as it is used to store a number of objects. We begin by introducing the container class *Bag*. Our specification for the class *Bag* is that it is a data structure into which objects can be inserted and from which objects can be deleted, like a bag of groceries. A bag can have multiple occurrences of the same element, but we do not care about the position of an element; nor do we care which element is removed when a delete operation is performed. We implement *Bag* by using a C++ array to hold its objects. Program 3.4 contains the class definition of a *Bag* of integers. The

default initial capacity of a bag is 10. When a *Bag* function is unable to perform its task, it is to throw an exception.

```

class Bag
{
public:
    Bag (int bagCapacity = 10); // constructor
    ~Bag();                     // destructor

    int Size() const;           // return number of elements in bag
    bool IsEmpty() const;       // return true if the bag is empty; false otherwise
    int Element() const;        // return an element that is in the bag

    void Push(const int);       // insert an integer into the bag
    void Pop();                 // delete an integer from the bag

private:
    int *array;
    int capacity;              // capacity of array
    int top;                   // array position of top element
};

```

Program 3.4: Definition of the class *Bag* containing integers

We implement insertion into the *Bag* by storing the element that is to be inserted into the first available position in the array. In case the array is full, its capacity is doubled. Deletion is implemented by deleting the element in the middle position of the array, and then moving all elements to its right, one position to the left. The function *Element* is implemented so as to return the element that will be deleted in case a *Pop* is done. Our implementation decisions for choosing the positions at which an element will be inserted or deleted as well as for choosing the element returned by *Element* are arbitrary. We could have just as easily decreed that elements should be inserted into the middle position, deleted from the last position, and that *Element* return the integer in the first position of the array. Program 3.5 contains our implementation of *Bag* operations.

Most of the operations of *Bag* need no explanation. We have declared member functions *Size()*, *IsEmpty()*, and *Element* as *inline* because of their small length (each contains only two lines of code). This eliminates the overhead of performing a function call. The three functions have the *const* attribute as they do not change the bag object upon which they operate.

132 Stacks and Queues

```

Bag::Bag (int bagCapacity): capacity (bagCapacity) {
    if (capacity < 1) throw "Capacity must be > 0";
    array = new int[capacity];
    top = -1;
}

Bag::~Bag() { delete [] array;}

inline int Bag::Size() const {return top+1;}

inline bool Bag::IsEmpty() const {return size == 0;}

inline int Bag::Element() const {
    if (!IsEmpty()) throw "Bag is empty";
    return array[0];
}

void Bag::Push(const int x) {
    if (capacity == top+1) ChangeSizeID(array, capacity, 2 * capacity);
    capacity *= 2;
    array[++top] = x;
}

void Bag::Pop() {
    if (!IsEmpty()) throw "Bag is empty, cannot delete";
    int deletePos = top / 2;
    copy(array + deletePos + 1, array + top + 1, array + deletePos);
    // compact array
    top--;
}

```

Program 3.5: Implementation of operations of Bag

As we have defined it, Bag can be used to store integers only. We would like to implement Bag using templates so that Bag can be used to store objects of any data type. Container classes are particularly suitable for implementation using templates because the algorithms for basic container class operations are usually independent of the type of objects that the container class contains. For example, the algorithm to add an element to Bag does not depend on the type of its elements. Program 3.6 contains a template class definition for Bag and

Program 3.7 contains the implementation of some of its operations. These are obtained from Programs 3.4 and 3.5, respectively, by prefixing the class definition of *Bag* and the definitions of all its member functions that are outside the class (in our example, all member functions are defined outside the class definition) with the statement:

```
template<class T>
```

Next, *int* is replaced by *T* when it refers to an object being stored in *Bag*. Finally, in all member function headers, *Bag::* is replaced by *Bag<T>::*. Notice, also, that the single input parameter to *Push* is made a constant reference. This is done because it is now possible that *T* represents a large object, and considerable overhead may be incurred if it is passed by value.

```
template <class T>
class Bag
{
public:
    Bag (int bagCapacity = 10);
    ~Bag();

    int Size() const;
    bool IsEmpty() const;
    T& Element() const;

    void Push(const T&);
    void Pop();

private:
    T *array;
    int capacity;
    int top;
};
```

Program 3.6: Definition of template class *Bag*

The following statements instantiate the template class *Bag* to *int* and *Rectangle*, respectively. So, *a* is a *Bag* of integers and *r* is a *Bag* of *Rectangles*.

```
Bag<int> a ;
Bag<Rectangle> r ;
```

```

template <class T>
Bag<T>::Bag (int bagCapacity): capacity ( bagCapacity ) {
    if (capacity < 1) throw "Capacity must be > 0";
    array = new T[capacity];
    top = -1;
}

template <class T>
Bag<T>::~Bag() {delete [] array;}

template <class T>
void Bag<T>::Push(const T& x) {
    if (capacity == top+1) {
        {
            ChangeSize/D(array, capacity, 2 * capacity);
            capacity *= 2;
        }
        array[++top] = x;
    }
}

template <class T>
void Bag<T>::Pop() {
    if (!IsEmpty()) throw "Bag is empty, cannot delete";
    int deletePos = top / 2;
    copy(array + deletePos + 1, array + top + 1, array + deletePos);
    // compact array
    array[top--] = T(); // destructor for T
}

```

Program 3.7: Implementation of some operations of Bag

3.2 THE STACK ABSTRACT DATA TYPE

We shall now study two data structures that are frequently used in computer programs. These data structures, the stack and the queue, are special cases of the more general data structure *ordered list* that we discussed in Section 2.3. In this section we begin by defining the ADT *Stack* and follow with its implementation. In Section 3.3 we look at the queue.

A *stack* is an ordered list in which insertions (also known as additions, puts, and pushes) and deletions (also known as removals and pops) are made at

one end called the *top*. Given a stack $S = (a_0, \dots, a_{n-1})$, we say that a_0 is the bottom element, a_{n-1} is the top element, and a_i is on top of element a_{i-1} , $0 < i < n$. The restrictions on the stack imply that if we add the elements A, B, C, D, E to the stack, in that order, then E is the first element we delete from the stack. Figure 3.1 illustrates this sequence of operations. Since the last element inserted into a stack is the first element removed, a stack is also known as a *Last-In-First-Out (LIFO)* list.



Figure 3.1: Inserting and deleting elements in a stack

Example 3.1 [System stack]: Before we discuss the stack ADT, we look at a special stack, called the system stack, that is used by a program at runtime to process function calls. Whenever a function is invoked, the program creates a structure, referred to as an *activation record* or a *stack frame*, and places it on top of the system stack. Initially, the activation record for the invoked function contains only a pointer to the previous stack frame and a return address. The previous stack frame pointer points to the stack frame of the invoking function; the return address contains the location of the statement to be executed after the function terminates. Since only one function executes at any given time, the function whose stack frame is on top of the system stack is chosen. If this function invokes another function, the local variables and the parameters of the invoking function are added to its stack frame. A new stack frame is then created for the invoked function and placed on top of the system stack. When this function terminates, its stack frame is removed and the processing of the invoking function, which is again on top of the stack, continues. A simple example illustrates this process.

Assume that we have a main function that invokes function $a1$. Figure 3.2(a) shows the system stack before $a1$ is invoked; Figure 3.2(b) shows the system stack after $a1$ has been invoked. Frame pointer fp is a pointer to the current stack frame. The system also maintains separately a stack pointer, sp , which we have not illustrated.

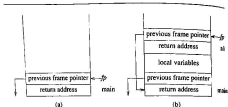


Figure 3.2: System stack after function call

Since all functions are stored similarly in the system stack, it makes no difference if the invoking function calls itself. That is, a recursive call requires no special strategy; the runtime program simply creates a new stack frame for each recursive call. However, recursion can consume a significant portion of the memory allocated to the system stack; it could consume the entire available memory. \square

Our discussion of the system stack suggests a basic set of operations, including insert an item, delete an item, and check for stack empty. These are given in the ADT specification (ADT 3.1). When an ADT function is unable to perform its task, it is to throw an exception.

The easiest way to implement the stack ADT is by using an array. `stack[i]`. The first, or bottom, element of the stack is stored in `stack[0]`, the second in `stack[1]`, and the i th in `stack[i-1]`. Associated with the array is a variable, `top`, which points to the top element in the stack. Initially, `top` is set to `-1` to denote an empty stack. This results in the following data member declarations and constructor definition of `Stack`:

```
private:
    T *stack;           // array for stack elements
    int top;            // array position of top element
    int capacity;       // capacity of stack array
```

```

template <class T>
class Stack
{ // A finite ordered list with zero or more elements.
public:
    Stack (int stackCapacity = 10);
        // Create an empty stack whose initial capacity is stackCapacity.

    bool IsEmpty() const;
        // If number of elements in the stack is 0, return true else return false.

    T& Top() const;
        // Return top element of stack.

    void Push (const T& item);
        // Insert item into the top of the stack.

    void Pop();
        // Delete the top element of the stack.
};

```

ADT 3.1: Abstract data type Stack

```

template <class T>
Stack<T>::Stack (int stackCapacity) : capacity (stackCapacity)
{
    if (capacity < 1) throw "Stack capacity must be > 0";
    stack = new T[capacity];
    top = -1;
}

```

The member functions *IsEmpty()* and *Top()* are implemented as follows:

```

template <class T>
inline bool Stack<T>::IsEmpty() const { return top == -1;}

template <class T>
inline T& Stack<T>::Top() const
{
    if (!IsEmpty()) throw "Stack is empty";
    return stack [top];
}

```

138 Stacks and Queues

The *Push* and *Pop* functions are shown in Programs 3.8 and 3.9.

```
template <class T>
void Stack<T>::Push (const T& x)
{ // Add x to the stack.
  if (top == capacity - 1)
  {
    ChangeSizeID(stack, capacity, 2*capacity);
    capacity *= 2;
  }
  stack[++top] = x;
}
```

Program 3.8: Adding to a stack

```
template <class T>
void Stack<T>::Pop()
{ // Delete top element from the stack.
  if (!IsEmpty()) throw "Stack is empty. Cannot delete.";
  stack[top--].~T(); // destructor for T
}
```

Program 3.9: Deleting from a stack

EXERCISES

1. Extend the stack ADT by adding functions to output a stack; split a stack into two stacks with one containing the bottom half elements and the second the remaining elements; and to combine two stacks into one by placing all elements of the second stack on top of those of the first stack. Write C++ code for your new functions.
2. Consider the railroad switching network given in Figure 3.3. Railroad cars numbered 1, 2, 3, ..., n are initially in the top right track segment (in this order, left to right). Railroad cars can be moved into the vertical track segment one at a time from either of the horizontal segments and then moved from the vertical segment to any one of the horizontal segments. The vertical segment operates as a stack as new cars enter at the top and cars depart the vertical segment from the top. For instance, if $n = 3$, we could move

car 1 into the vertical segment, move 2 in, move 3 in, and then take the cars out producing the new order 3, 2, 1. For $n = 3$ and 4 what are the possible permutations of the cars that can be obtained? Are any permutations not possible?

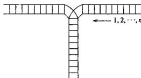


Figure 3.3: Railroad switching network

3.3 THE QUEUE ABSTRACT DATA TYPE

A queue is an ordered list in which insertions (also called additions, puts, and pushes) and deletions (also called removals and pops) take place at different ends. The end at which new elements are added is called the *rear*, and that from which old elements are deleted is called the *front*. The restrictions on a queue imply that if we insert A , B , C , D , and E in that order, then A is the first element deleted from the queue. Figure 3.4 illustrates this sequence of events. Since the first element inserted into a queue is the first element removed, queues are also known as *First-In-First-Out (FIFO)* lists. The ADT specification of the queue appears in ADT 3.2. When an ADT function is unable to perform its task, it is to throw an exception.

Analogous to the representation of a stack in Section 3.2, we may use an array `queue[]` with the first (or front) element of the queue in `queue[0]`, the next in `queue[1]`, and so on. Figure 3.5(a) shows a 3-element queue represented in this way and Figure 3.5(b) shows the queue after an element has been deleted. Figure 3.5(c) shows the result of adding an element to the queue of Figure 3.5(b). Since a delete or pop removes the front element, which is in `queue[0]`, each delete requires us to shift the remaining elements to the left. Hence, it takes $\Theta(n)$ time to pop an element from a queue that has n elements; an insertion or push, on the other hand, can be done in $\Theta(1)$ time exclusive of the time for any array resizing



Figure 3.4: Inserting and deleting elements in a queue

```

template <class T>
class Queue
{ // A finite ordered list with zero or more elements.
public:
    Queue(int queueCapacity = 10);
    // Create an empty queue whose initial capacity is queueCapacity

    bool IsEmpty() const;
    // If number of elements in the queue is 0, return true else return false.

    T& Front() const;
    // Return the element at the front of the queue.

    T& Rear() const;
    // Return the element at the rear of the queue.

    void Push(const T& item);
    // Insert item at the rear of the queue.

    void Pop();
    // Delete the front element of the queue.
};

```

ADT 3.2: Abstract data type Queue

that may be needed.

To pop an element in $\Theta(1)$ time, we must relax the requirement that `queue[0]` contain the front element of the queue. With this relaxation, we use a variable, *front*, to keep track of the location of the front element. The queue

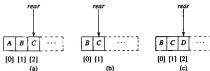


Figure 3.5: Queues represented with front element in queue [0]

elements are in queue $[front], \dots, queue[rear]$. Figure 3.6 shows three queues represented in this way. The queue of Figure 3.6(b) results from the deletion of the front element of the queue of Figure 3.6(a) and that of Figure 3.6(c) results from adding an element to the queue of Figure 3.6(b).

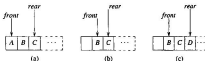


Figure 3.6: Queues represented with front element in queue $[front]$

Suppose that the capacity of the array *queue* is *capacity*. The representation of Figure 3.6 runs into a problem when *rear* equals *capacity* - 1 and *front* > 0 as in Figure 3.7(a). How do we add an element to this configuration without increasing array capacity? One possibility is to shift all elements to the left end of the queue (as in Figure 3.7(b)) and create space at the right end. This shifting takes time proportional to the size (i.e., number of elements) of the queue.

The worst-case add and delete times (assuming no array resizing is needed) become $\Theta(1)$ when we permit the queue to wrap around the end of the array. At this time it is convenient to think of the array positions as arranged in a circle

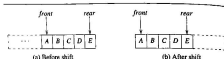


Figure 3.7: Shifting queue elements to the left

(Figure 3.8) rather than in a straight line (Figure 3.7). In Figure 3.8, we have changed the convention for the variable *front*. This variable now points one position counterclockwise from the location of the front element in the queue. The convention for *rear* is unchanged. This change simplifies the codes slightly.

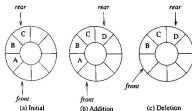


Figure 3.8: Circular queue

When the array is viewed as a circle, each array position has a next and a previous position. The position next to position $\text{capacity} - 1$ is 0, and the position that precedes 0 is $\text{capacity} - 1$. When the queue *rear* is at $\text{capacity} - 1$, the next element is put into position 0. To work with a circular queue, we must be able to move the variables *front* and *rear* from their current position to the next

position (clockwise). This may be done using code such as

```
if (rear == capacity - 1) rear = 0;
else rear++;
```

Using the modulus operator, which computes remainders, this code is equivalent to $(rear+1) \% \text{capacity}$. With our conventions for *front* and *rear*, we see that the front element of the queue is located one position clockwise from *front* and the rear element is at position *rear*.

Suppose we implement the ADT *Queue* using an array in a circular fashion. We may use the following data member declarations and constructor.

```
private:
    T *queue;      // array for queue elements
    int front,     // one counterclockwise from front
        rear,     // array position of rear element
        capacity; // capacity of queue array

template <class T>
Queue<T>::Queue (int queueCapacity) : capacity (queueCapacity)
{
    if (capacity < 1) throw "Queue capacity must be > 0";
    queue = new T[capacity];
    front = rear = 0;
}
```

To determine a suitable test for an empty queue, we experiment with the queues of Figure 3.8. To delete an element, we advance *front* one position clockwise and to add an element, we advance *rear* one position clockwise and insert at the new position. If we perform 3 deletions from the queue of Figure 3.8(c) in this fashion, we will see that the queue becomes empty and that *front* = *rear*. When we do 5 additions to the queue of Figure 3.8(b), the queue becomes full and *front* = *rear*. So, we cannot distinguish between an empty and a full queue. To avoid the resulting confusion, we shall increase the capacity of a queue just before it becomes full. Consequently, *front* == *rear* iff the queue is empty. The member functions *IsEmpty()*, *Front()*, and *Rear()* are implemented as follows:

```
template <class T>
inline bool Queue<T>::IsEmpty() {return front == rear;}

template <class T>
inline T& Queue<T>::Front()
{
```


144 Stacks and Queues

```

    if (!IsEmpty()) throw "Queue is empty. No front element";
    return queue[(i/front + 1) % capacity];
}

template <class T>
inline T& Queue<T>::Rear()
{
    if (!IsEmpty()) throw "Queue is empty. No rear element";
    return queue[rear];
}

```

Program 3.10 gives the code to add an element to a queue. This code uses custom code to double the capacity of *queue* when necessary. To see the need for custom code to double capacity, consider the full queue of Figure 3.9(a). This figure shows a queue with seven elements in an array whose capacity is 8. To visualize array doubling when a circular queue is used, it is better to flatten out the array as in Figure 3.9(b). Figure 3.9(c) shows the array after array doubling by *ChangeSize1D* (Program 3.3).

```

template <class T>
void Queue<T>::Push(const& x)
// Add x at rear of queue.
{
    if ((rear + 1) % capacity == front)
        // queue full, double capacity
        // code to double queue capacity comes here
    }

    rear = (rear + 1) % capacity;
    queue[rear] = x;
}

```

Program 3.10: Adding to a queue

To get a proper circular queue configuration, we must slide the elements in the right segment (i.e., elements *A* and *B*) to the right end of the array as in Figure 3.10(d). The array doubling and the slide to the right together copy at most $2 \times \text{capacity} - 1$ elements. The number of elements copied can be limited to capacity $- 1$ by customizing the array doubling code so as to obtain the configuration of Figure 3.10(e). This configuration may be obtained as follows:

- (1) Create a new array *newQueue* of twice the capacity,

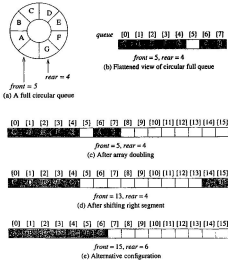


Figure 3.9: Doubling queue capacity

146 Stacks and Queues

- (2) Copy the second segment (i.e., the elements `queue[front+1]` through `queue[capacity-1]`) to positions in `newQueue` beginning at 0.
- (3) Copy the first segment (i.e., the elements `queue[0]` through `queue[rear]`) to positions in `newQueue` beginning at `capacity-front-1`.

The code of Program 3.11 obtains the configuration of Figure 3.10(e). Program 3.12 gives the code to delete or pop an element from a queue. Its complexity is $O(1)$.

```
// allocate an array with twice the capacity
T* newQueue = new T[2*capacity];

// copy from queue to newQueue
int start = (front + 1) % capacity;
if (start < 2)
    // no wrap around
    copy(queue + start, queue + start + capacity - 1, newQueue);
else
    // queue wraps around
    copy(queue + start, queue + capacity, newQueue);
    copy(queue, queue + rear + 1, newQueue + capacity - start);
}

// switch to newQueue
front = 2*capacity - 1;
rear = capacity - 2;
capacity *= 2;
delete [] queue;
queue = newQueue;
```

Program 3.11: Doubling queue capacity

One way to use all positions in the array `queue` is to use an additional variable, `lastOp`, to record the last operation performed on the queue. This variable is initialized to "Pop." Following each addition, it is set to "Push" and following each deletion, it is set to "Pop." Now when `front == rear`, we can determine whether the queue is empty or full by examining the value of `lastOp`. If `lastOp` is "Push," then the queue is full. Otherwise, it is empty. The use of the variable `lastOp` as described above does, however, slow down the `Push` and `Pop` functions. Since the `Push` and `Pop` functions will be used many times in any problem

```

template <class T>
void Queue<T>::Pop()
// Delete: front element from queue.
    if (!IsEmpty()) throw "Queue is empty. Cannot delete.";
    front = (front + 1) % capacity;
    queue[front].~T(); // destructor for T
}

```

Program 3.12: Deleting from a queue

involving queues, the loss of one queue position will be more than made up for by the reduction in computing time. Hence, we favor the implementations of Programs 3.10 and 3.12 over those that result from the use of the variable *LenOp*.

EXERCISES

1. Rewrite functions *Push* and *Pop* (Programs 3.10 and 3.12) using the variable *lenOp* as discussed in this section. The queue should now be able to hold up to capacity elements. The complexity of each of your functions should be $O(1)$ (exclusive of the time taken to double queue capacity when needed).
2. To the class *Queue*, add functions to return the size and capacity of a queue.
3. To the class *Queue*, add a function to split a queue into two queues. The first queue is to contain every other element beginning with the first; the second queue contains the remaining elements. The relative order of queue elements is unchanged. What is the complexity of your function?
4. To the class *Queue*, add a function to merge two queues into one by alternately taking elements from each queue. The relative order of queue elements is unchanged. What is the complexity of your function?
5. A double-ended queue (deque) is a linear list in which additions and deletions may be made at either end. Obtain a data representation mapping a deque into a one-dimensional array. Write C++ template functions to add and delete elements from either end of the deque and also to return an element from either end.
6. A linear list is being maintained circularly in an array with *front* and *rear* set up as for circular queues.
 - (a) Obtain a formula in terms of the array capacity, *front*, and *rear*, for

148 Stacks and Queues

the number of elements in the list.

- (b) Write a C++ template function to delete the k th element in the list.
- (c) Write a C++ template function to insert an element y immediately after the k th element. Use array doubling in case you need to increase the capacity of the element array.
- (d) Develop a complete template class *List* that includes the functions of (b) and (c) as well as a constructor and destructor. Your class also should include functions to return the k th element in the list and the size of the list. An *IsEmpty* function should be included as well. Test all functions.

What is the time complexity of your functions for (b) and (c)?

3.4 SUBTYPING AND INHERITANCE IN C++

Inheritance is used to express subtype relationships between ADTs. This is also referred to as the IS-A relationship. We say that a data object of Type *B* IS-A data object of Type *A* if *B* is more specialized than *A* or *A* is more general than *B*; that is, all *B*s are *A*s, but not all *A*s are *B*s; or the set of all objects of Type *B* is a subset of the set of all objects of Type *A*. For example, *Chair* IS-A *Furniture*, *Living IS-A Mammal*, *Rectangle* IS-A *Polygon*.

Consider the relationship between *Bag* and *Stack*. A bag is simply a data structure into which elements can be inserted and from which elements can be deleted. A stack is also a data structure into which elements can be inserted and from which elements can be deleted. However, a stack is more specialized in that it requires that elements be deleted in last-in first-out order. So, a stack can be used instead of a bag, but a bag cannot be used for an application that requires a stack. So, *Stack* IS-A *Bag* or *Stack* is a subtype of *Bag*. The IS-A relationship is a fundamental, conceptual relationship between the specifications of two ADTs. The relationship is not affected by the implementation of either ADT. So, *Stack* IS-A *Bag* is true, even if their implementations are changed (as long as their ADT specifications are met).

C++ provides a mechanism to express the IS-A relationship called *public inheritance*. Program 3.13 shows how public inheritance is implemented in C++. The declaration `public Bag` appended to class name `Stack` implies that *Stack* publicly inherits from *Bag*. Here, *Bag*, representing the more general ADT is called the *base class*, while *Stack*, representing the more specialized ADT is called the *derived class*. You will notice that some member functions are preceded by the keyword `virtual`. We will discuss this in the next chapter.

An important consequence of inheritance is that the derived class (*Stack*) inherits

```

class Bag
{
public:
    Bag (int bagCapacity = 10);
    virtual ~Bag();

    virtual int Size() const;
    virtual bool IsEmpty() const;
    virtual int Element() const;

    virtual void Push(const int);
    virtual void Pop();
protected:
    int *array;
    int capacity;
    int top;
};

class Stack : public Bag
{
public:
    Stack (int stackCapacity = 10);
    ~Stack();
    int Top() const;
    void Pop();
};

```

Program 3.13: Definition of *Bag* and *Stack*

all the members (data and functions) of the base class (*Bag*). However, only the non-private members of the base class are accessible to the derived class. This means that any members in *Bag* that are protected or public may be accessed by *Stack*. So, *array*, *capacity*, and *top* may be accessed by *Stack* even though they are not defined in the class definition of *Stack*. Similarly, *Size*, *IsEmpty*, *Element*, and *Push* are accessible to *Stack* even though they are not defined in *Stack*.

Another important consequence of public inheritance is that inherited public and protected members of the base class have the same level of access in the derived class as they did in the base class. So all three data members are protected members of *Stack*. The functions *Size*, *IsEmpty*, *Element*, and *Push* are all public members of *Stack*.

150 Stacks and Queues

The member functions inherited by *Stack* from *Bag* have the same prototypes. This is a reuse of the interface of a base class. Notice also that the only operation that has a different implementation in *Bag* and *Stack* is *Pop*. The other operations all have identical implementations. C++ allows us to reuse the base-class implementation of an operation in a derived class. In the event that the implementation of a derived class operation must be different from the base class implementation, it is also possible to override the base class implementation. In our example, only *Pop* is redefined in the derived class. (The constructor and destructor cannot be inherited and therefore must also be redefined in the derived class.) The implementations of all *Bag* operations are assumed to be identical to those in Program 3.3. In Program 3.14, we show the member functions of *Stack* that need to be reimplemented. The implementations of all other functions are inherited from *Bag* and hence do not have to be reimplemented.

```
Stack::Stack(int stackCapacity): Bag(stackCapacity) { }  
// Constructor for Stack calls constructor for Bag.  
  
Stack::~Stack() { }  
// Destructor for Bag is automatically called when Stack  
// is destroyed. This ensures that array is deleted.  
  
int Stack::Top() const  
{  
    if (!IsEmpty()) throw "Stack is empty."  
    return array[top];  
}  
  
void Stack::Pop()  
{  
    if (!IsEmpty()) throw "Stack is empty. Cannot delete."  
    top--;  
}
```

Program 3.14: Implementation of *Stack* operations

The following code fragment illustrates how inheritance works:

```
Bag b(3); // uses Bag constructor to create array of capacity 3  
Stack s(3); // uses Stack constructor to create array of capacity 3
```

```
b.Push(1); b.Push(2); b.Push(3);
// use Bag::Push.
```

```
s.Push(1); s.Push(2); s.Push(3);
// Stack::Push not defined, so use Bag::Push.
```

```
b.Pop(); // uses Bag::Pop, which calls Bag::IsEmpty
s.Pop();
// uses Stack::Pop, which calls Bag::IsEmpty because IsEmpty
// has not been redefined in Stack.
```

All operations on base class objects (such as *b* in the example) work in exactly the same way they would have, had there been no inheritance. Operations on derived class objects (such as *s* in the example) work differently: if an operation is defined in *Stack*, then that definition is used. If an operation is not defined in *Stack*, then the operation defined in the class that *Stack* inherits from (*Bag* in our example) is used. After executing the above code, *b* contains 1 and 3, while *s* contains 1 and 2.

Note that the following also are permissible:

```
s.Size(); // uses Bag::Size
```

```
s.Element(); // uses Bag::Element
```

If we do not wish to provide access to certain base class functions, then we must redefine these functions in the derived class. So, for example, we may redefine *Element* in *Stack* so as to throw an exception.

The *Queue* ADT is also a subtype of *Bag*. It is more specialized than *Bag* because it requires that elements be deleted in first-in first-out order. Therefore, *Queue* can also be represented as a derived class of *Bag*. However, there is less similarity between the implementations of *Bag* and *Queue* than there is between *Bag* and *Stack*. Like *Bag* and *Stack*, *Queue* uses an array, but unlike *Bag* and *Stack*, *Queue* requires the data members *front* and *rear*. The implementations of the operations *IsEmpty*, *Push*, and *Pop* are different from those of *Bag*. This means that a larger number of operations need to be redefined in *Queue* than were redefined in *Stack*. Even though there is not much reuse of implementation, there is still reuse of interfaces because the redefined functions of *Queue* have the same prototypes as the original functions in *Bag*. The exercises explore a full implementation of *Queue* as a derived class of *Bag*.

EXERCISES

1. Implement *Stack* as a publicly derived class of *Bag* using templates.
2. Implement *Queue* as a publicly derived class of *Bag* using templates.
3. A *double-ended queue* (*deque*) is a linear list in which additions and deletions may be made at either end. Implement the class *Deque* as a publicly derived class of *Queue*. The class *Deque* must have public functions (either via inheritance from *Queue* or by direct implementation in *Deque*) to add and delete elements from either end of the deque and also to remove an element from either end. The complexity of each function (exclusive of array doubling) should be $\Theta(1)$.
4. For each of the following pairs state whether they are linked by the IS-A relationship. Justify your answer.
 - (a) *Rectangle* and *Trapezium*.
 - (b) *Rectangle* and *Circle*.
 - (c) *Lion* and *Tiger*.
 - (d) *Stack* and *Ordered List*.
 - (e) *Queue* and *Ordered List*.

3.5 A MAZING PROBLEM

The rat in a maze experiment is a classical one from experimental psychology. A rat (or mouse) is placed through the door of a large box without a top. Walls are set up so that movements in most directions are obstructed. The rat is carefully observed by several scientists as it makes its way through the maze until it eventually reaches the exit. There is only one way out, but at the end is a nice chunk of cheese. The idea is to run the experiment repeatedly until the rat will zip through the maze without taking a single false path. The trials yield its learning curve.

We can write a computer program for getting through a maze, and it will probably not be any smarter than the rat on its first try through. It may take many false paths before finding the right one. But the computer can remember the correct path far better than the rat. On its second try it should be able to go right to the end with no false paths taken, so there is no sense re-running the program. Why don't you sit down and try to write this program yourself before you read on and look at our solution. Keep track of how many times you have to go back and correct something. This may give you an idea of your own learning curve as we re-run the experiment throughout the book.

Let us represent the maze by a two-dimensional array, *maze*[*i*][*j*], where

$1 \leq i \leq m$ and $1 \leq j \leq p$. A value of 1 implies a blocked path, and a 0 means one can walk right on through. We assume that the rat starts at `maze[1][1]`, and the exit is at `maze[m][p]`. An example is given in Figure 3.11.

entrance	0	1	0	0	0	1	1	0	0	0	1	1	1	1	1
	1	0	0	0	1	1	0	1	1	1	0	0	1	1	1
	0	1	1	0	0	0	0	1	1	1	1	0	0	1	1
	1	1	0	1	1	1	1	0	1	1	0	1	1	0	0
	1	1	0	1	0	0	1	0	1	1	1	1	1	1	1
	0	0	1	1	0	1	1	1	0	1	0	0	1	0	1
	0	0	1	1	0	1	1	1	0	1	0	0	1	0	1
	0	1	1	1	1	0	0	1	1	1	1	1	1	1	1
	0	0	1	1	0	1	1	0	1	1	1	1	1	0	1
	1	1	0	0	0	1	1	0	1	1	0	0	0	0	0
	0	0	1	1	1	1	1	0	0	0	1	1	1	1	0
	0	1	0	0	1	1	1	1	0	1	1	1	1	0	
exit															

Figure 3.11: An example maze (can you find a path?)

With the maze represented as a two-dimensional array, the location of the rat in the maze can at any time be described by the row, i , and the column, j , of its position. Now let us consider the possible moves the rat can make from a point $[i][j]$ in the maze. Figure 3.12 shows the possible moves from any point $[i][j]$. The position $[i][j]$ is marked by an X. If all the surrounding squares have a 0, then the rat can choose any of these eight squares as its next position. We call these eight directions by the names of the points on a compass: north, northeast, east, southeast, south, southwest, west, and northwest, or N, NE, E, SE, S, SW, W, and NW.

We must be careful here because not every position has eight neighbors. If $[i][j]$ is on a border where either $i = 1$ or m , or $j = 1$ or p , then less than eight, and possibly only three, neighbors exist. To avoid checking for border conditions, we surround the maze by a border of ones. The array will therefore be declared as `maze[m+2][p+2]`. Another device that will simplify the problem is to predefine the possible directions to move in a table, `move`, as in Figure 3.13. The data types needed are

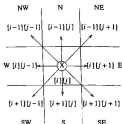


Figure 3.12: Allowable moves

```

struct offsets
{
    int a, b ;
};

enum directions { N, NE, E, SE, S, SW, W, NW };
offsets move[8] ;

```

If we are at position (i, j) in the maze and we wish to find the position (g, h) that is southwest of us, then we set

$$g = i + \text{move}[\text{SW}].a; \quad h = j + \text{move}[\text{SW}].b;$$

For example, if we are at position $(3)[4]$, then the position to the southwest is given by $(3 + 1 = 4) [4 + (-1) = 3]$.

As we move through the maze, we may have the chance to go in several directions. Not knowing which one to choose, we pick one but save our current position and the direction of the last move in a list. This way, if we have taken a false path, we can return and try another direction. With each new location we will examine the possibilities, starting from the north and looking clockwise. Finally, in order to prevent us from going down the same path twice, we use

<i>g</i>	<i>move</i> [<i>g</i>]. <i>a</i>	<i>move</i> [<i>g</i>]. <i>b</i>
N	-1	0
NE	-1	1
E	0	1
SE	1	1
S	1	0
SW	1	-1
W	0	-1
NW	-1	-1

Figure 3.13: Table of moves

another array, *mark*[*m* + 2][*p* + 2], which is initially zero. *mark*[*r*][*j*] is set to 1 once we arrive at that position. We assume *maze*[*m*][*p*] = 0, since otherwise there is no path to the exit. Program 3.15 is a first pass at an algorithm.

This is not a C++ program, and yet it describes the essential processing without too much detail. The use of indentation for delineating important blocks of code plus the use of C++ reserved words make the looping and conditional tests transparent.

What remains to be pinned down? Using the three arrays *maze*, *mark*, and *move*, we need only specify how to represent the list of triples (*i*, *j*, *dir*). Since the algorithm calls for removing first the most recently entered triple, this list should be a stack. To avoid doubling array capacity during a stack insertion, we need to choose a sufficiently large initial capacity for the stack. Since each position in the maze is visited at most once, at most *m**p* elements can be placed into the stack. Thus, an initial capacity of *m**p* is a safe but somewhat conservative value to use for the initial capacity of the stack. The maze of Figure 3.14 has only one path from entrance to exit. It has $\lfloor m/2 \rfloor (p-3) + 2$ positions. Thus, *m**p* is not too crude a bound. We are now ready to give a precise path construction function (Program 3.16).

The arrays *maze*, *mark*, and *move*, are assumed global to *Push*. Further, *stack* is defined to be a stack of *Items* where the struct *Items* is defined as

```
struct Items {
    int x, y, dir;
};
```

The struct *Items* has a constructor (not shown) that sets the values of *x*, *y* and *dir*. The operator << is overloaded for both *Stack* and *Items* as shown in Program

```

initialize list to the maze entrance coordinates and direction east;
while (list is not empty)
{
    (i,j,dir) = coordinates and direction from end of list;
    delete last element of list;
    while (there are more moves from (i,j))
    {
        (g,h) = coordinates of next move;
        if ((g == m) && (h == p)) success;
        if ( (!move [g][h]) // legal move
            && (!mark [g][h]) // haven't been here before
        )
        {
            mark [g][h] = 1;
            dir = next direction to try;
            add (i,j,dir) to end of list;
            (i,j,dir) = (g,h,N);
        }
    }
}
cout << "No path in maze." << endl;

```

Program 3.15: First pass at finding a path through a maze

3.17. We assume that the operator << has been given access to the private data members of *Stack* through the use of the friend declaration.

Analysis of Path: Now, what can we say about the computing time of the function *Path*? It is interesting that even though the problem is easy to grasp, it is difficult to make any but the most trivial statement about the computing time. The reason for this is that the number of iterations of the main **while** loop is entirely dependent upon the given maze. What we can say is that each new position [*i*][*j*] that is visited gets marked, so paths are never taken twice. There are at most eight iterations of the inner **while** loop for each marked position. Each iteration of the inner **while** loop takes a fixed amount of time, $O(1)$, and if the number of zeros in *maze* is z , then at most z positions can get marked. Since z is bounded above by np , the computing time is $O(np)$. (In actual experiments, however, the rat may be inspired by the watching psychologists and the invigorating odor from the cheese at the exit. It might reach its goal by examining far fewer paths than those examined by function *Push*. This may happen despite the fact that the rat has no pencil and only a very limited mental stack. It is

entrance	0	0	0	0	0	0	0	1
	1	1	1	1	1	1	1	0
	1	0	0	0	0	0	0	1
	0	1	1	1	1	1	1	1
	1	0	0	0	0	0	0	1
	1	1	1	1	1	1	1	0
	1	0	0	0	0	0	0	1
	0	1	1	1	1	1	1	1
	1	0	0	0	0	0	0	0
	1	0	0	0	0	0	0	0
exit								

Figure 3.14: A maze with a long path

difficult to incorporate the effect of the cheese odor and the cheering of the psychologists into a computer program.) Note that the array *mark* can be eliminated altogether and *maze[g][h]* changed to 1 instead of setting *mark[g][h]* to 1, but this will destroy the original maze. □

EXERCISES

- Find a path through the maze of Figure 3.11.
 - Trace out the action of function *Path* (Program 3.16) on the maze of Figure 3.11. Compare this to your own attempt in (a).
- What is the maximum path length from start to finish for any maze of dimensions $m \times p$?
- Write and test recursive version of *Path*. What is the time complexity of your recursive version?

3.6 EVALUATION OF EXPRESSIONS

3.6.1 Expressions

When pioneering computer scientists conceived the idea of higher-level programming languages, they were faced with many hurdles. One of these was how to generate machine-language instructions to evaluate an arithmetic expression. An assignment statement such as

```

void Path(const int m, const int p)
// Output a path (if any) in the maze; maze[0][i] = maze[m+1][i] =

    // start at (1,1)
    mark[1][1] = 1;
    Stack<temp> stack(temp);
    from temp(1, 1, E);
    // set temp.x, temp.y, and temp.dir
    stack.Push(temp);
    while (!stack.IsEmpty())
    // stack not empty
        temp = stack.Top();
        stack.Pop(); // unstack
        int i = temp.x; int j = temp.y; int d = temp.dir;
        while (d < 8) // move forward
        {
            int g = i + move[d].x; int h = j + move[d].y;
            if ((g == m) && (h == p)) { // reached exit
                // output path
                cout << stack;
                cout << i << " " << j << endl; // last two squares on the path
                cout << m << " " << p << endl;
                return;
            }
            if (!maze[g][h] && (!mark[g][h])) { // new position
                mark[g][h] = 1;
                temp.x = g; temp.y = h; temp.dir = d+1;
                stack.Push(temp); // stack it
                i = g; j = h; d = N; // move to (g,h)
            }
            else d++; // try next direction
        }
    }
    cout << "No path in maze." << endl;
}

```

Program 3.16: Finding a path through a maze

```

template <class T>
ostream& operator<<(ostream& os, Stack<T>& s)
{
    os << "top = " << s.top << endl;
    for (int i = 0; i <= s.top; i++)
        os << i << ": " << s.stack[i] << endl;
    return os;
}

ostream& operator<<(ostream& os, Item& item)
{
    return os << item.x << ", " << item.y << ", " << item.dir;
}

```

Program 3.17: Overloading <<

$$X = A/B - C + D * E - A * C$$

might have several meanings, and even if it were uniquely defined, say by a full use of parentheses, it still seemed a formidable task to generate a correct instruction sequence. Fortunately the solution we have today is both elegant and simple. Moreover, it is so simple that this aspect of compiler writing is really one of the more minor issues.

An expression is made up of operands, operators, and delimiters. The expression above has five operands: *A*, *B*, *C*, *D*, and *E*. Though these are all one-letter variables, operands can be any legal variable name or constant in our programming language. In any expression, the values that variables take must be consistent with the operations performed on them. These operations are described by the operators. In most programming languages there are several kinds of operators that correspond to the different kinds of data a variable can hold. First, there are the basic arithmetic operators: plus, minus, times, and divide (+, -, *, /). Other arithmetic operators include unary minus, and %. A second class is the relational operators: <, <=, ==, <>, >=, and >. These are usually defined to work for arithmetic operands, but they can just as easily work for character string data. ("CAT" is less than "DOG" since it precedes "DOG" in alphabetical order.) The result of an expression that contains relational operators is one of two constants: true or false. Such an expression is called Boolean, named after the mathematician George Boole, the father of symbolic logic. There also may be logical operators such as &&, ||, and !.

The first problem with understanding the meaning of an expression is to decide in what order the operations are carried out. This means that every

160 Stacks and Queues

language must uniquely define such an order. For instance, if $A = 4$, $B = C = 2$, $D = E = 3$, then in the above equation we might want X to be assigned the value

$$\begin{aligned} & ((4/2) - 2) + (3 + 3) - (4 * 2) \\ &= 0 + 9 - 8 \\ &= 1 \end{aligned}$$

However, the true intention of the programmer might have been to assign X the value

$$\begin{aligned} & (4/(2 - 2 + 3)) * (3 - 4) * 2 \\ &= (4/3) * (-1) * 2 \\ &= -2.66666666 \end{aligned}$$

Of course, the programmer could specify the latter order of evaluation by using parentheses:

$$X = ((A/(B - C + D)) * (E - A) * C$$

To fix the order of evaluation, we assign to each operator a priority. Then within any pair of parentheses we understand that operators with the highest priority will be evaluated first. A set of sample priorities from C++ is given in Figure 3.15. The highest priority is 1.

priority	operator
1	unary minus, !
2	*, /, %
3	+, -
4	<, <=, >=, >
5	=, !=
6	&&
7	

Figure 3.15: Priority of operators in C++

Unary minus and the logical not (!) have top priority, followed by *, /, and %. When we have an expression where two adjacent operators have the same priority, we need a rule to tell us which one to perform first. For example, do we want the value of $A/B * C$ to be understood as $(A/B) * C$ or $A/(B * C)$? Convince yourself that there will be a difference by trying $A = B = C = 2$. The C++ rule is that for all priorities, evaluation of operators of the same priority will proceed left to right. So, $A/B * C$ will be evaluated as $(A/B) * C$. Remember that by

using parentheses we can override these rules, as expressions are always evaluated with the innermost parenthesized expression first.

Now that we have specified priorities and rules for breaking ties we know how $X = A/B - C + D * E - A * C$ will be evaluated, namely, as

$$X = ((A/B) - C) + (D * E) - (A * C)$$

3.6.2 Postfix Notation

How can a compiler accept an expression and produce correct code? The answer is given by reworking the expression into a form we call *postfix notation*. If e is an expression with operators and operands, the conventional way of writing e is called *infix*, because the operators come in-between the operands. (Unary operators precede their operand.) The *postfix* form of an expression calls for each operator to appear *after* its operands. For example,

infix $A * B / C$ has postfix $AB * C /$

If we study the postfix form of $A * B / C$, we see that the multiplication comes immediately after its two operands A and B . Now imagine that $A * B$ is computed and stored in T . Then we have the division operator, $/$, coming immediately after its two operands T and C .

Let us look at our previous example

infix: $A/B - C + D * E - A * C$
 postfix: $AB / C - DE * + AC - +$

and trace out the meaning of the postfix expression.

Suppose that every time we compute a value, we store it in the temporary location T_i , $i \geq 1$. If we read the postfix expression left to right, the first operation is division. The two operands that precede this are A and B . So, the result of A/B is stored in T_1 , and the postfix expression is modified as in Figure 3.16. This figure also gives the remaining sequence of operations. The result is stored in T_6 . Notice that if we had parenthesized the expression, this would change the postfix only if the order of normal evaluation were altered. Thus, $(A/B) - C + (D * E) - A * C$ will have the same postfix form as the previous expression without parentheses. But $(A/B) - (C + D) * (E - A) * C$ will have the postfix form $AB / C D + EA - * C * -$.

What are the virtues of postfix notation that enable easy evaluation of expressions? To begin with, the need for parentheses is eliminated. Second, the priority of the operators is no longer relevant. The expression may be evaluated by making a left to right scan, stacking operands, and evaluating operators using

operation	postfix
$T_1 = A/B$	$T_1 C - DE * + AC * -$
$T_2 = T_1 - C$	$T_2 DE * + AC * -$
$T_3 = D * E$	$T_3 T_1 + AC * -$
$T_4 = T_2 + T_3$	$T_4 AC * -$
$T_5 = A * C$	$T_4 T_5 -$
$T_6 = T_4 - T_5$	T_6

Figure 3.16: Postfix evaluation

as operands the correct number from the stack and finally placing the result onto the stack (see Program 3.18). This evaluation process is much simpler than attempting direct evaluation from infix notation.

```

void Eval(Expression e)
// Evaluate the postfix expression e. It is assumed that the last token (a token
// N is either an operator, operand, or '#' in e is '#'. A function NextToken is
// used to get the next token from e. The function uses the stack stack
    Stack<Token> stack; // initialize stack
    for (Token x = NextToken(e); x != '#'; x = NextToken(e))
        if (x is an operand) stack.Push(x) // add to stack
        else // N operator
            remove the correct number of operands for operator x from stack;
            perform the operation x and store the result (if any) onto the stack;
    }

```

Program 3.18: Evaluating postfix expressions

3.6.3 Infix to Postfix

To see how to devise an algorithm for translating from infix to postfix, note that the order of the operands in both forms is the same. In fact, it is simple to describe an algorithm for producing postfix from infix:

- (1) Fully parenthesize the expression.

- (2) Move all operators so that they replace their corresponding right parentheses.
- (3) Delete all parentheses.

For example, $A/B - C + D * E - A * C$, when fully parenthesized, yields

$$((((A/B) - C) + (D * E)) - (A * C))$$

The arcs join an operator and its corresponding right parenthesis. Steps 2 and 3 yield

$$AB/C - DE * + AC * -$$

Since the order of the operands is the same in infix and postfix, when we scan an expression for the first time, we can form the postfix by immediately passing any operands to the output. To handle the operators, we store them in a stack until it is time to pass them to the output.

For example, since we want $A + B * C$ to yield $ABC * +$, our algorithm should perform the following sequence of stacking (these stacks will grow to the right):

next token	stack	output
none	empty	none
A	empty	A
+	+	A
B	+	AB

At this point the algorithm must determine if $*$ gets placed on top of the stack or if the $+$ gets taken off. Since $*$ has higher priority, we should stack $*$, producing

+	++	AB
C	++	ABC

Now the input expression is exhausted, so we output all remaining operators in the stack to get

$$ABC * +$$

For another example, $A + (B + C) * D$ has the postfix form $ABC * + D +$, and so the

164 Stacks and Queues

algorithm should behave as

next token	stack	output
none	empty	none
A	empty	A
*	*	A
(*(A
B	*(AB
+	*(+	AB
C	*(+	ABC

At this point we want to unstack down to the corresponding left parenthesis and then delete the left and right parentheses. This gives us

)	*	ABC +
*	*	ABC ++
D	*	ABC ++ D
done	empty	ABC ++ D +

These examples motivate a priority-based scheme for stacking and unstacking operators. The left parenthesis complicates things, since when it is not in the stack, it behaves as an operator with high priority, whereas once it gets in, it behaves as one with low priority (no operator other than the matching right parenthesis should cause it to get unstacked). We establish two priorities for operators: *isp* (in-stack priority) and *icp* (in-coming priority). The *isp* and *icp* of all operators in Figure 3.15 remain unchanged. In addition, we assume that *isp('(')* returns 8, *icp('(')* returns 0, and *isp('*')* returns 8. These priorities result in the following rule: *Operators are taken out of the stack as long as their in-stack priority is numerically less than or equal to the in-coming priority of the new operator.* Our function to transform from infix to postfix is given in Program 3.19.

Analysis of *Postfix*: The function makes only a left-to-right pass across the input. The time spent on each operand is $O(1)$. Each operator is stacked and unstacked at most once. Hence, the time spent on each operator is also $O(1)$. So, the complexity of *Postfix* is $O(n)$, where n is the number of tokens in the expression. \square

```

void Postfix (Expression e)
// Output the postfix form of the infix expression e. NextToken
// is as in function Eval (Program 3.18). It is assumed that
// the last token in e is '#'. Also, '#' is used at the bottom of the stack
    Stack<Token> stack; // initialize stack
    stack.Push('#');
    for (Token x = NextToken(e); x != '#'; x = NextToken(e))
        if (x is an operand) cout << x;
        else if (x == '(')
            // unstack until ')'
            for (; stack.Top() != ')'; stack.Pop())
                cout << stack.Top();
            stack.Pop(); // unstack ')'
        else { // x is an operator
            for (; isop(stack.Top()) <= isop(x); stack.Pop())
                cout << stack.Top();
            stack.Push(x);
        }

    // end of expression; empty the stack
    for (; !stack.IsEmpty(); cout << stack.Top(), stack.Pop());
    cout << endl;
}

```

Program 3.19: Converting from infix to postfix form

EXERCISES

- Write the postfix form of the following expressions:
 - $A * B + C$
 - $-A + B - C + D$
 - $A * -B + C$
 - $(A + B) * D + E \wedge F + A * D + C$
 - $A \ \&\& \ B \ \|\ C \ \& \ (E > F)$ (assuming C++ precedence)
 - $\neg (A \ \&\& \ \neg ((B < C) \vee (C > D))) \ \|\ (C < E)$

166 Stacks and Queues

- Use the priorities of Figure 3.15 together with those for '(' and ')' to answer the following:
 - In function *Postfix* (Program 3.19), what is the maximum number of elements that can be on the stack at any time if the input expression e has n operators and delimiters?
 - What is the answer to (a) if e has n operators and the depth of nesting of parentheses is at most 6?
- Another expression form that is easy to evaluate and is parenthesis-free is known as *prefix*. In this way of writing expressions, the operators precede their operands. For example:

infix	prefix
$A * B / C$	$/ * A B C$
$A / B - C + D * E - A * C$	$- + - / A B C * D E * A C$
$A * (B + C) / D - G$	$- / * A + B C D G$

Notice that the order of operands is not changed in going from infix to prefix.

- What is the prefix form of the expressions in Exercise 1?
- Write a C++ function to evaluate a prefix expression e . (Hint: Scan e right to left and assume that the leftmost token of e is '#'.)
- Write a C++ function to transform an infix expression e into its prefix equivalent. Assume that the input expression e begins with a '#' and that the prefix expression should begin with a '#'.)

What is the time complexity of your functions for (b) and (c)? How much space is needed by each of these functions?

- Write a C++ function to transform from prefix to postfix. Carefully state any assumptions you make regarding the input. How much time and space does your function take?
- Do the preceding exercise, but this time for a transformation from postfix to prefix.
- Write a C++ function to generate fully parenthesized infix expressions from their postfix form. What is the complexity (time and space) of your function?
- Do the preceding exercise starting from prefix form.

3.7 ADDITIONAL EXERCISES

1. [*Programming Project*] [Landweber] People have spent so much time playing card games of solitaire that the gambling casinos are now capitalizing on this human weakness. A form of solitaire is described below. Your assignment is to write a computer program to play the game, thus freeing hours of time for people to return to more useful endeavors.

To begin the game, 28 cards are dealt into seven piles. The leftmost pile has one card, the next two cards, and so forth, up to seven cards in the rightmost pile. Only the uppermost card of each of the seven piles is turned face up. The cards are dealt left to right, one card to each pile, dealing to one less pile each time, and turning the first card in each round face up. On the topmost face-up card of each pile you may build in descending sequence red on black or black on red. For example, on the 9 of spades you may place either the 8 of diamonds or the 8 of hearts. All face-up cards on a pile are moved as a unit and may be placed on another pile according to the bottommost face-up card. For example, the 7 of clubs on the 8 of hearts may be moved as a unit onto the 9 of clubs or the 9 of spades.

Whenever a face-down card is uncovered, it is turned face up. If one pile is removed completely, a face-up king may be moved from a pile (together with all cards above it) or the top of the waste pile (see below) into the vacated space. There are four output piles, one for each suit, and the object of the game is to get as many cards as possible into the output piles. Each time an ace appears at the top of a pile or the top of the stack, it is moved into the appropriate output pile. Cards are added to the output piles in sequence, the suit for each pile being determined by the ace on the bottom.

From the rest of the deck, called the stock, cards are turned up one by one and placed face up on a waste pile. You may always play cards off the top of the waste pile, but only one at a time. Begin by moving a card from the stock to the top of the waste pile. If there is ever more than one possible play to be made, the following order must be observed:

- (a) Move a card from the top of a playing pile or from the top of the waste pile to an output pile. If the waste pile becomes empty, move a card from the stock to the waste pile.
- (b) Move a card from the top of the waste pile to the leftmost playing pile to which it can be moved. If the waste pile becomes empty, move a card from the stock to the waste pile.
- (c) Find the leftmost playing pile that can be moved and place it on top

of the leftmost playing pile to which it can be moved.

- (d) Try (a), (b), and (c) in sequence, restarting with (a) whenever a move is made.
- (e) If no move is made via (a) through (d), move a card from the stock to the waste pile and retry (a).

Only the topmost card of the playing piles or the waste pile may be played to an output pile. Once played on an output pile, a card may not be withdrawn to help elsewhere. The game is over when either all the cards have been played to the output, or the stock pile has been exhausted and no more cards can be moved.

When played for money, the player pays the house \$52 at the beginning and wins \$5 for every card played to the output piles. Write your program so that it will play several games and determine your net winnings. Use a random number generator to shuffle the deck. Output a complete record of two games in easily understood form. Include as output the number of games played and the net winnings (+ or -).

2. [Programming Project] [Landweber] We want to simulate an airport landing and takeoff pattern. The airport has three runways, runway 1, runway 2, and runway 3. There are four landing holding patterns, two for each of the first two runways. Arriving planes will enter one of the holding pattern queues, where the queues are to be as close in size as possible. When a plane enters a holding queue, it is assigned an integer *id* number and an integer giving the number of time units the plane can remain in the queue before it must land (because of low fuel level). There is also a queue for takeoffs for each of the three runways. Planes arriving in a takeoff queue are also assigned an integer *id*. The takeoff queues should be kept approximately the same size.

At each time, up to three planes may arrive at the landing queues and up to three planes may arrive at the takeoff queues. Each runway can handle one takeoff or landing at each time slot. Runway 3 is to be used for takeoffs except when a plane is low on fuel. At each time unit, planes in either landing queue whose air time has reached zero must be given priority over other landings and takeoffs. If only one plane is in this category, runway 3 is to be used. If more than one, then the other runways are also used (at each time, at most three planes can be serviced in this way).

Use successive even (odd) integers for *id*'s of planes arriving at takeoff (landing) queues. At each time unit assume that arriving planes are entered into queues before takeoffs or landings occur. Try to design your algorithm so that neither landing nor takeoff queues grow excessively. However, arriving planes must be placed at the ends of queues. Queues cannot be

reordered.

The output should clearly indicate what occurs at each time unit. Periodically output (a) the contents of each queue; (b) the average takeoff waiting time; (c) the average landing waiting time; (d) the average flying time remaining on landing; and (e) the number of planes landing with no fuel reserve. (b) and (c) are for planes that have taken off or landed, respectively. The output should be self-explanatory and easy to understand (and uncluttered).

The input can be from a terminal, a file, or it can be generated by a random number generator. For each time unit the input consists of the number of planes arriving at take-off queues, the number of planes arriving at landing queues, and the remaining flying times for each plane arriving at a landing queue.

CHAPTER 4

Linked Lists

4.1 SINGLY LINKED LISTS AND CHAINS

In the previous chapters, we studied the representation of simple data structures using an array and a sequential mapping. These representations had the property that successive nodes of the data object were stored a fixed distance apart. Thus, (1) if the element a_{ij} of a table was stored at location L_{ij} , then $a_{i,j+1}$ was at the location $L_{ij} + 1$; (2) if the i th element in a queue was at location L_i , then the $(i + 1)$ th element was at location $(L_i + 1) \% n$ for the circular representation; (3) if the topmost node of a stack was at location L_T , then the node beneath it was at location $L_T - 1$, and so on. These sequential storage schemes proved adequate for the tasks we wished to perform (accessing an arbitrary node in a table, insertion or deletion of stack and queue elements). However, when a sequential mapping is used for ordered lists, operations such as insertion and deletion of arbitrary elements become expensive. For example, consider the following list of three-letter English words ending in AT:

(BAT, CAT, EAT, FAT, HAT, JAT, LAT, MAT, OAT, PAT, RAT, SAT, VAT, WAT)

To make this list more complete we naturally want to add the word GAT, which means gun or revolver. If we are using an array and a sequential mapping to keep this list, then the insertion of GAT will require us to move elements already in the list either one location higher or lower. We must move either HAT, IAT, LAT, \dots , WAT or BAT, CAT, EAT, and FAT. If we have to do many such insertions into the middle, neither alternative is attractive because of the amount of data movement. Excessive data movement also is required for deletions. Suppose we decide to remove the word LAT, which refers to the Latvian monetary unit. Then again, we have to move many elements so as to maintain the sequential representation of the list.

An elegant solution to this problem of data movement in *sequential representations* is achieved by using *linked representations*. Unlike a sequential representation, in which successive items of a list are located a fixed distance apart, in a linked representation these items may be placed anywhere in memory. In other words, in a sequential representation the order of elements is the same as in the ordered list, whereas in a linked representation these two sequences need not be the same. To access list elements in the correct order, with each element we store the address or location of the next element in that list. Thus, associated with each data item in a linked representation is a pointer or link to the next item. In general, a linked list is comprised of nodes; each node has zero or more data fields and one or more link or pointer fields.

Figure 4.1 shows how some of the elements in our list of three-letter words may be represented in memory by using pointers. The elements of the list are stored in a one-dimensional array called *data*, but the elements no longer occur in sequential order, BAT before CAT before EAT, and so on. Instead we relax this restriction and allow them to appear anywhere in the array and in any order. To remind us of the real order, a second array, *link*, is added. The values in this array are pointers to elements in the *data* array. For any i , *data*[i] and *link*[i] together comprise a node. Since the list starts at *data*[8] = BAT, let us set a variable *first* = 8. *link*[8] has the value 3, which means it points to *data*[3], which contains CAT. Since *link*[3] = 4, the next element, EAT, in the list is in *data*[4]. The element after EAT is in *data*[*link*[4]]. By continuing in this way we can list all the words in the proper order. We recognize that we have come to the end of our ordered list when *link* equals zero. To ensure that a link of zero always signifies the end of a list, we do not use position zero of *data* to store a list element.

It is customary to draw linked lists as an ordered sequence of nodes with links being represented by arrows, as in Figure 4.2. Notice that we do not explicitly put in the values of the pointers but simply draw arrows to indicate they are there. The arrows reinforce in our own mind the facts that (1) the nodes do not actually reside in sequential locations and (2) the actual locations of nodes are immaterial. Therefore, when we write a program that works with lists, we do not

	data	link
1	HAT	15
2		
3	CAT	4
4	EAT	9
5		
6		
7	WAT	0
8	BAT	3
9	FAT	1
10		
11	VAT	7
	.	.
	.	.
	.	.

Figure 4.1: Nonsequential list representation.

look for a specific address except when we test for zero. The linked structures of Figures 4.1 and 4.2 are called *singly linked lists* or *chains*. In a *singly linked list*, each node has exactly one pointer field. A *chain* is a singly linked list that is comprised of zero or more nodes. When the number of nodes is zero, the chain is empty. The nodes of a non-empty chain are ordered so that the first node links to the second node; the second to the third; and so on. The last node of a chain has a 0 link.



Figure 4.2: Usual way to draw a linked list.

Let us now see why it is easier to make insertions and deletions at arbitrary

positions using a linked list rather than a sequential list. To insert the data item GAT between FAT and HAT, the following steps are adequate:

- (1) Get a node *a* that is currently unused.
- (2) Set the *data* field of *a* to GAT.
- (3) Set the *link* field of *a* to point to the node after FAT, which contains HAT.
- (4) Set the *link* field of the node containing FAT to *a*.

Figure 4.3(a) shows how the arrays *data* and *link* will be changed after we insert GAT. Figure 4.3(b) shows how we can draw the insertion using our arrow notation. Dashed arrows are new ones. The important thing to notice is that when we insert GAT, we do not have to move any elements that are already in the list. We have overcome the need to move data at the expense of the storage needed for the field *link*. Usually, this penalty is not too severe. When each list element is large, significant time is saved by not having to move elements during an insert or delete.

Now suppose we want to delete GAT from the list. All we need to do is find the element that immediately precedes GAT, which is FAT, and set *link*[9] to the position of HAT which is 1. Again, there is no need to move the data around. Even though the link of GAT still contains a pointer to HAT, GAT is no longer in the list as it cannot be reached by starting at the first element of list and following links (see Figure 4.4).

4.2 REPRESENTING CHAINS IN C++

In the following subsections, we shall see the basic techniques used to implement chains in C++. In the next section, we generalize the representation using templates.

4.2.1 Defining A Node in C++

To define the structure of a node, we need to know the type of each of its fields. The *data* field in the example of the previous section is simply a three-character string, and the *link* field is a pointer to another node.

If the type of the node in our earlier example is denoted by *ThreeLetterNode*, then the data type *ThreeLetterNode* may be defined as a class as follows:

```
class ThreeLetterNode {
private:
    char data[3];
    ThreeLetterNode *link;
};
```

Example 4.1 defines two node structures that can be combined with each

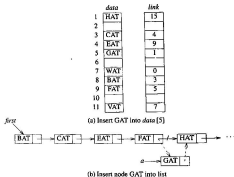


Figure 4.3: Inserting into a linked list

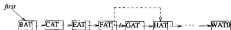


Figure 4.4: Delete GAT

other to form a complex list structure.

Example 4.1: The class definitions

```
class NodeA {
private:
    int data1;
    char data2;
    float data3;
    NodeA *linka;
    NodeB *linkb;
};

class NodeB {
private:
    int data;
    NodeB *link;
};
```

define *NodeA* to consist of three data fields and two link fields, whereas nodes of type *NodeB* consist of one data field and one link field. Note that the first data member of nodes of type *NodeA* must point to nodes of type *NodeB*. Figure 4.5 shows a linked list that has one node of each type. □

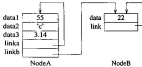


Figure 4.5: Illustration of the node structures *NodeA* and *NodeB*

4.3.2 Designing a Chain Class in C++

We have seen how to represent a single node in C++. Now, we shall design a C++ representation for a chain of nodes. The basic elements of our design are especially important because they apply not just to chains, but also to other linked data structures that we shall study later. As is typical with any design process, we shall explore different alternatives, some of which will turn out to be infeasible or undesirable.

Design Attempt 1: The variable *first*, in the *ThreeLetterNode* example, is declared as

```
ThreeLetterNode *first;
```

The data members of the node pointed to by *first* may be referenced in the following way:

```
first->data, first->link
```

and the components of *data* are referenced as

```
first->data[0], first->data[1], first->data[2]
```

This is shown diagrammatically in Figure 4.6.



Figure 4.6: Referencing the data members of a node

There is, however, a flaw in our implementation of *ThreeLetterNode*. Notice that the data members *data* and *link* are declared as **private**. This is in keeping with the data encapsulation principles discussed in Chapter 1. The consequence of declaring the data members *data* and *link* as **private** is that the expressions *first*→*link*, *first*→*data*[0], *first*→*data*[1], and *first*→*data*[2] will all result in compile-time errors because private data members cannot be accessed

from outside the class!

Design Attempt 2: How do we overcome the drawback of our first attempt? Making the data members public would do the trick, but that violates data encapsulation. A solution might be to define public member functions *GetLink()*, *SetLink()*, *GetData()*, and *SetData()* on *ThreeLetterNode* which are used to indirectly access the data members. This is not a good solution, however, because this solution allows one to read and change those data members from anywhere in the program. The ideal solution would only permit those functions that perform list manipulation operations (like inserting a node into or deleting a node from a chain) access to the data members of *ThreeLetterNode*.

Design Attempt 3: Let us try to tackle this data structure design problem from a different perspective: the data structure we are interested in implementing is a chain of three-letter words. This suggests that our program should contain a class corresponding to the entire chain. Let us call this class *ThreeLetterChain*. This class should contain member functions that carry out list manipulation operations such as insert and delete. Thus, we will be using a composite of two classes, *ThreeLetterNode* and *ThreeLetterChain*, to implement our chain. The conceptual relationship between the two classes may be characterized by the following statement: an object of the class *ThreeLetterChain* consists of zero or more objects of the class *ThreeLetterNode*; or, *ThreeLetterChain* HAS-A *ThreeLetterNode*.

Definition: We say that a data object of type *A* HAS-A data object of type *B* if *A* conceptually contains *B* or *B* is a part of *A*. For example, *Computer* HAS-A *Processor*, *Book* HAS-A *Page*, and so on. HAS-A relationships between ADTs are usually expressed by making the contained class a member of the containing class. This is possible when the type *A* object contains a fixed number of type *B* objects. □

Figure 4.7 shows the conceptual relationship between the two classes. Figure 4.7 suggests that an object of class *ThreeLetterChain* physically contains many objects of class *ThreeLetterNode*; that is objects of *ThreeLetterNode* are declared as data members of *ThreeLetterChain*. But, the number of nodes in a chain is not a constant; on the contrary, it varies with each insert or delete operation performed on the list. So, it is impossible to know in advance the number of *ThreeLetterNodes* to include in *ThreeLetterChain*. For this reason, the *ThreeLetterChain* class is defined so that it only contains the access pointer, *first*, which points to the first node in the list. Figure 4.8 shows the actual relationship between *ThreeLetterChain* and *ThreeLetterNode* objects. It shows that *ThreeLetterNode* objects are not physically contained inside *ThreeLetterChain*. The only

ThreeLetterChain

Figure 4.7: Conceptual relationship between *ThreeLetterChain* and *ThreeLetterNode*

data member contained in *ThreeLetterChain* is the pointer *first*.

ThreeLetterChain

Figure 4.8: Actual relationship between *ThreeLetterChain* and *ThreeLetterNode*

We are now finally ready to propose a solution to our dilemma of how to define the class *ThreeLetterNode* so that only list manipulation operations have access to the data members of its nodes. This is achieved by declaring *ThreeLetterChain* to be a *friend* of *ThreeLetterNode*. Only member functions of *ThreeLetterChain* and *ThreeLetterNode* can now access the private data members of *ThreeLetterNode*. The class definitions of *ThreeLetterChain* and *ThreeLetterNode* are shown in Program 4.1.

Nested Classes: An alternative method to represent a chain in C++ is to use nested classes, where one class is defined inside the definition of another class. Here, class *ThreeLetterNode* is *defined* inside the private portion of the definition of class *ThreeLetterChain*, as shown in Program 4.2. This ensures that

```
class ThreeLetterChain; // forward declaration
```

```
class ThreeLetterNode {
friend class ThreeLetterChain;
private:
    char data[3];
    ThreeLetterNode *link;
};
```

```
class ThreeLetterChain {
public:
    // Chain manipulation operations
    .
    .
private:
    ThreeLetterNode *first;
};
```

Program 4.1: Composite classes

ThreeLetterNode objects cannot be accessed outside class *ThreeLetterChain*. Notice that the data members of *ThreeLetterNode* are public. This ensures that they can be accessed by member functions of *ThreeLetterChain*. Using nested classes achieves the same effect as our previous approach, which uses composite classes. We use the composite class approach in the rest of the text. One of our reasons for preferring composite classes over nested classes is that a single node class can be used by many different classes (so long as each is a friend of the node class).

4.2.3 Pointer Manipulation in C++

Nodes of a predefined type may be created using the C++ command *new*. If *f* is of type *ThreeLetterNode** then following the call *f = new ThreeLetterNode*, **f* denotes the node of type *ThreeLetterNode* that is created. Similarly, if *a* and *b* are of type *NodeA** and *NodeB**, respectively, (Example 4.1) then following the execution of the statements

```

class ThreeLetterChain {
public:
    // Chain Manipulation operations
    .
    .
private:
    class ThreeLetterNode { // nested class
    public:
        char data[3];
        ThreeLetterNode *link;
    };
    ThreeLetterNode *first;
};

```

Program 4.2: Nested classes

```

a = new NodeA;
b = new NodeB;

```

*a, and *b, respectively, denote the nodes of type *NodeA* and *NodeB* that are created. These nodes may be deleted in the following way:

```
delete f; delete a; delete b;
```

C++ also allows pointer variables to be assigned the null pointer constant 0 (or NULL). This constant is used to denote a pointer that points to no node (e.g., the *link* data member in the last node of Figure 4.3(b)) or an empty list (as in *first* = 0). Addition of integers to pointer variables is permitted in C++ (but is generally only used in the context of arrays). Thus, if *x* is a pointer variable, the expression *x* + 1 is valid, but may have no logical meaning. Two pointer variables of the same type may also be compared to see if both point to the same node. Thus, if *x* and *y* are pointer variables of the same type, then the expressions *x* == *y*, *x* != *y*, *x* == 0, *x* != 0 are all valid. The effect of the assignments *x* = *y* and **x* = **y* on the initial configuration of Figure 4.9(a) is given in Figure 4.9(b) and (c). When a pointer is output in C++ using *cout*, the location in memory that the pointer is addressing is output.

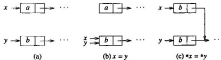


Figure 4.9: Effect of pointer assignments

4.2.4 Chain Manipulation Operations

In this subsection, we look at three functions that manipulate chains. These functions use the classes *ChainNode* and *Chain*, where *ChainNode* is defined as:

```
class ChainNode {
friend class Chain;
public:
    ChainNode(int element = 0, ChainNode* next = 0)
        // 0 is the default value for element and next
        {data = element; link = next;}
private:
    int data;
    ChainNode *link;
};
```

The access pointer *first*, which points to the first node in the chain, is a private data member of *Chain*. It is assumed that the three functions we are about to develop are declared in the public segment of the class definition of *Chain*.

Example 4.2: Function *Chain::Create2* (Program 4.3) creates a chain with two nodes. The data field of the first node is set to 10 and that of the second to 20. Function *Create2* uses the constructor for *ChainNode* to initialize the fields of the two newly created nodes. The resulting list structure is shown in Figure 4.10. □

Example 4.3: Let *first* be a pointer to the first node of a chain. *first* == 0 iff the chain is empty (i.e., there are no nodes on the chain). Let *x* be a pointer to some arbitrary node in the chain. Program 4.4 inserts a node with data field 50

```

void Chain::Create2()
{
    // create and set fields of second node
    ChainNode* second = new ChainNode(20,0);

    // create and set fields of first node
    first = new ChainNode(10, second);
}

```

Program 4.3: Creating a two-node list



Figure 4.10c: A two-node list

following the node pointed to by x except when the chain is empty. Once again, the constructor for `ChainNode` is used to initialize the fields of the new node. The resulting chain structures for the two cases $first == 0$ and $first != 0$ are shown in Figure 4.11. □

```

void Chain::Insert50(ChainNode *x)
{
    if (first)
        // insert after x
        x->first = new ChainNode(50, x->first);
    else
        // insert into empty list
        first = new ChainNode(50);
}

```

Program 4.4: Inserting a node

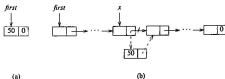


Figure 4.11: Inserting into an empty and nonempty list

Example 4.4: Let *first* and *x* be as in Example 4.3. Let *y* point to the node (if any) that precedes *x*, and let *y* = 0 iff *x* == *first*. Function `Chain::Delete` (Program 4.5) deletes node *x* from the chain. □

```
void Chain::Delete(ChainNode *x, ChainNode *y)
{
    if (x == first) first = first->link;
    else y->link = x->link;
    delete x;
}
```

Program 4.5: Deleting a node

EXERCISES

The following exercises are based on the definitions of Section 4.2.4. All functions are to be implemented as member functions of *Chain* and are therefore assumed to have access to the data members of *ChainNode*.

1. Write a C++ function *length* to count the number of nodes in a chain. What is the time complexity of your function?
2. Let *x* be a pointer to an arbitrary node in a chain. Write a C++ function to delete this node from the chain. If *x* == *first*, then *first* should be reset to point to the new first node in the chain. What is the time complexity of

your function?

- Write a C++ function to delete every other node beginning with node *first* (i.e., the first, third, fifth, and so on nodes of the chain are deleted). What is the time complexity of your function?
- Let $x = (x_1, x_2, \dots, x_n)$ and $y = (y_1, y_2, \dots, y_m)$ be two chains. Write a C++ function to merge the two chains together to obtain the chain $z = (z_1, z_2, \dots, z_{m+n})$ if $m \leq n$ and $z = (z_1, z_2, \dots, z_n)$ if $m > n$. Following the merge, x and y should represent empty chains because each node initially in x or y is now in z . No additional nodes may be used. What is the time complexity of your function?
- Let $x = (x_1, x_2, \dots, x_n)$ and $y = (y_1, y_2, \dots, y_m)$ be two chains. Assume that in each chain, the nodes are in nondecreasing order of their data-field values. Write a C++ function to merge the two chains to obtain a new chain z in which the nodes are also in this order. Following the merge, x and y should represent empty chains because each node initially in x or y is now in z . No additional nodes may be used. What is the time complexity of your function?
- It is possible to traverse a chain in both directions (i.e., left to right and a restricted right-to-left traversal) by reversing the links during the left-to-right traversal. A possible configuration under this scheme is given in Figure 4.12. Assume that l and r are data members of *Chain*.



Figure 4.12: Possible configuration for a chain traversed in both directions

The pointer r points to the node currently being examined and l to the node on its left. Note that all nodes to the left of r have their links reversed.

- Write a C++ function to move pointer r , n nodes to the right from any given position (l, r). If n is greater than the number of nodes to the right, set r to 0 and l to the rightmost node in the chain.
- Write a C++ function to move r , n nodes to the left from any given position (l, r). If n is greater than the number of nodes to the left, set l

to 0 and *r* to the leftmost node in the chain.

4.3 THE TEMPLATE CLASS CHAIN

The cost of developing software systems is significant. A contributor is the large number of programmer-hours required to develop, test, and maintain software. Thus, there is a need for software development strategies that reduce the number of person-hours spent without sacrificing the quality of the software. One technique for this is software reuse. The basic principle of software reusability is that when we initially design and develop software, we must try and do so in a manner that makes it possible to reuse the software in the future. This typically requires a greater investment of time when the software is first developed but pays dividends in terms of time saved when the software is reused.

In this section, we shall enhance the chain class of the previous section so that it becomes more reusable. If we are later required to develop software for an application that uses chains, we will be able to reuse our enhanced chain class.

4.3.1 Implementing Chains with Templates

We have already seen that the template mechanism can be used to make a container class more reusable. A chain is clearly a container class, and is therefore a good candidate for implementation with templates. Program 4.6 contains the template definition of our enhanced chain class. Notice that `Chain<T>` has been made a friend of `ChainNode<T>`. This means that the members of `ChainNode<int>` can be accessed by members of `Chain<int>`, but not by members of, say, `Chain<float>`. Similarly, notice that `first` is declared to be a pointer to an object of type `ChainNode<T>`. This means that an object of type `Chain<int>`, say, only consists of nodes of type `ChainNode<int>`.

An empty chain of integers *inlist* would be defined as:

```
Chain<int> inlist;
```

4.3.2 Chain Iterators

An iterator is an object that is used to access the elements of a container class one by one. The following discussion motivates the need for iterators of a container class. Consider the following operations that one might wish to perform on a container class *C*, all of whose elements are integers.

```

template <class T> class Chain; // forward declaration

template <class T>
class ChainNode {
friend class Chain <T>;
private:
    T data;
    ChainNode <T> *link;
};

template <class T>
class Chain {
public:
    Chain() {first = 0;} // constructor initializing first to 0
    // Chain manipulation operations
    .
    .
private:
    ChainNode <T> *first;
};

```

Program 4.6: Template definition of chains

- (1) Output all integers in C .
- (2) Obtain the maximum, minimum, mean, median, or mode of all integers in C .
- (3) Obtain the sum, product, or sum of squares of all integers in C .
- (4) Obtain all integers in C that satisfy some property P (for example, P could be *is a positive integer*, *is a square of an integer*, etc.).
- (5) Obtain the integer x from C such that, for some function f , $f(x)$ is maximum.

Notice that the solutions to each of these problems require you to examine all elements of the container class. The pseudo-code for these problems typically takes the following form:

```

initialization step;
for each item in C
{
    currentItem = current item of C;
    do something with currentItem;
}
postprocessing step;

```

For example, to find the maximum of all elements in the container class, we would use the following pseudo-code:

```

1 int x = std::numeric_limits<int>::min(); // initialization, must include <limits>
2 for each item in C
3 {
4     currentItem = current item of C;
5     x = max(currentItem, x);           // do something
6 }
7 return x;                             // postprocessing step

```

Program 4.7: Pseudo-code for computing maximum element

The C++ implementation of lines 2 and 4 depends on the container class being used. For an array *a* of size *n*, the implementation of these lines is:

```

2 for (int i = 0; i < n; i++)
4 currentItem = a[i];

```

For a nonempty chain of integers, the implementation of lines 2 and 4 is:

```

2 for ( ChainNode<int> *ptr = first; ptr != 0; ptr = ptr->link)
4 currentItem = ptr->data;

```

Operations such as finding the max element have to be implemented as member functions of the particular container class as these operations access private data members of the container class. There are some drawbacks to this approach. For example, consider the container class *Chain* <*T*>.

(1) Since *Chain* <*T*> is a template class, all of its operations should preferably be independent of the type of object to which *T* is instantiated. However, operations that are meaningful for one instantiation of *T* may not be meaningful for

188 Linked Lists

another instantiation. For example, computing the sum of all elements in a chain makes sense when $T = \text{int}$, but not when $T = \text{Rectangle}$.

(2) The number of member functions of $\text{Chain}<T>$ can become quite large, making the class definition rather unwieldy. A class definition should be as compact as possible so that it is easier for a class user to understand. Moreover, classes are often part of a class library designed by one programmer or programming team. It is not feasible for the class designer to predict all the operations required by a particular user of the class. If a user requires an operation that is not supported by the class, he or she would be forced to add that operation to the container class definition. This is not considered to be a good practice since the resulting class definition may not be consistent with the objectives with which the original class was designed.

(3) Even if it is acceptable for the user to add member functions to a class, he or she would have to learn how to sequence through the elements of the container class, which would entail learning how the container class is implemented.

These arguments suggest that container classes be equipped with iterators that provide systematic access to the elements of the object. Users can employ these iterators to implement their own functions depending upon the particular application. Typically, an iterator is implemented as a nested class of the container class.

C++ Iterators

In C++, an iterator is a pointer to an element of an object (for example, a pointer to an element of an array or chain). As the name suggests, an iterator permits you to go (or iterate) through the elements of the object one by one. Program 4.8 shows how to use a pointer (or iterator) y to an array element to iterate through the array's elements. The datatype of the pointer y is int^* , which indicates that y points to elements of type int . In the `for` loop header, y is initialized to point to the first element, $x[0]$, in the array $x[]$ (technically, the variable x is a pointer to the first element of the array). The expression $y++$ increments the pointer so that it advances to the next element of the array. Similarly, $x + 3$ is a pointer 3 positions from x ; that is, it points one position past the last element $x[2]$ of the array. So, in the `for` loop of Program 4.8, y iterates through elements in the range $[x, x + 3)$. The expression $*y$ dereferences y so as to get the element pointed to by y . The program outputs $x[0:2]$.

The following code is equivalent to the `for` loop of Program 4.8.

```
void main()
{
    int x[3] = {0, 1, 2};

    // use a pointer y to iterate through the array x
    for (int* y = x; y != x + 3; y++)
        cout << *y << " ";
    cout << endl;
}
```

Program 4.8: Using an array iterator

```
for (int i = 0; i != 3; i++)
    cout << x[i] << " ";
```

Although you may find this code more transparent than that of Program 4.8, the code of Program 4.8 is generalized easily to output the elements of any object for which an iterator is defined. For example, the code

```
for (Iterator i = start; i != end; i++)
    cout << *i << " ";
```

outputs all elements in the range $[start, end)$. In this code *Iterator* is the datatype of the iterator, *start* is the iterator value for the first element in the range and *end* is the value the iterator has when incremented one past the last element to be output.

The concept of an iterator is fundamental to writing generic code in C++. Program 4.9 gives a possible code for the STL *copy* function. This code may be used to copy elements of any object that has an iterator for which the operators *!=*, and *++* (postincrement) as well as the dereferencing operator (***) have been defined. Different generic codes we write require our iterator to have different capabilities. For example, the STL *copy_backward* function requires us to decrement the value of the iterator.

To simplify iterator development and categorization of generic iterator-based codes, the C++ STL defines five categories of iterators: input, output, forward, bidirectional and random access. All iterators support the equality operators *==* and *!=* as well as the dereference operator ***. Input iterators additionally provide read access to the elements pointed at and support the pre- and post-increment operator *++*. Output iterators provide write access to the elements and also permit iterator advancement via the *++* operator. Forward iterators may be

```
template <class Iterator>
void copy(Iterator start, Iterator end, Iterator to)
{if copy from [start, end) to [to, to + end - start)
  while (start != end)
    {*to = *start; start++; to++;}
}
```

Program 4.9: Possible code for STL *copy* function

advanced using the increment operator `++` while bidirectional iterators may be incremented as well as decremented (`--`). Random access iterators are the most general. They permit pointer jumps by arbitrary amounts as well as pointer arithmetic. C++ array iterators such as *y* in Program 4.8 are random access iterators.

A Forward Iterator for Chain

We may implement a forward iterator class for Chain as in Program 4.10. Our implementation requires that *ChainIterator* be a public nested member class of *Chain*.

Additionally, we add the following public member functions to *Chain*.

```
ChainIterator begin() {return ChainIterator(first);}
ChainIterator end() {return ChainIterator(0);}
```

which respectively return iterators initialized to the first node of a list and one past the last node. We may initialize an iterator object *yi* to the start of a chain of integers *y* using the statement

```
Chain<int>::ChainIterator yi = y.begin();
```

and we may sum the elements in this chain using the STL algorithm *accumulate* and the statement

```
sum = accumulate(y.begin(), y.end(), 0);
```

4.3.3 Chain Operations

Perhaps the most important decision when designing a reusable class is choosing which operations to include. It is important to provide enough operations so that the class can be used in many applications. It is also important not to include too many operations because this will make the class bulky. Examples of operations that would be included in most reusable classes are constructors (including

```

class ChainIterator {
public:
    // typedefs required by C++ for a forward iterator omitted

    // constructor
    ChainIterator(ChainNode<T>* startNode = 0)
        {current = startNode;}

    // dereferencing operators
    T& operator*() const {return current->data;}
    T* operator->() const {return &current->data;}

    // increment
    ChainIterator& operator++() // postincrement
        {current = current->link; return *this;}
    ChainIterator operator++(int) // postincrement
        {
            ChainIterator old = *this;
            current = current->link;
            return old;
        }

    // equality testing
    bool operator!=(const ChainIterator right) const
        {return current != right.current;}
    bool operator==(const ChainIterator right) const
        {return current == right.current;}
private:
    ChainNode<T>* current;
};

```

Program 4.10: A forward iterator for Chain

default and copy constructors), a destructor, an assignment operator (`operator=`), the test-for-equality operator (`operator==`), and operators to input and output a class object (obtained by overloading `operator>>` and `operator<<`, respectively). Other operators may also be overloaded depending on the class.

A chain class should provide functions to insert and delete elements. There could be a number of variations under each of these categories. For example, there could be an insertion function that inserts an element at the front of a chain,

192 Linked Lists

another that inserts at the end, as in Program 4.11, and yet another that inserts after the i th element of the chain. To insert efficiently at the end of a chain, we add a private data member *last* to the class *Chain*<*T*>. This data member points to the last node in the chain.

```
template <class T>
void Chain<T>::InsertBack(const T& x)
{
    if (!first) { // nonempty chain
        last->link = new ChainNode<T>(x);
        last = last->link;
    }
    else first = last = new ChainNode<T>(x);
}
```

Program 4.11: Inserting at the back of a list

Program 4.12 concatenates two chains. Like the function *InsertBack*, the function *Concatenate* assumes that *Chain* has been augmented with the private data member *last*. The complexity of this function is $O(1)$.

```
template <class T>
void Chain<T>::Concatenate(Chain<T>& b)
// b is concatenated to the end of *this.
{
    if (!first) { last->link = b.first; last = b.last; }
    else { first = b.first; last = b.last; }
    b.first = b.last = 0;
}
```

Program 4.12: Concatenating two chains

Another useful function is one that reverses the order of elements in a chain (Program 4.13). This function is especially interesting because it does an “in-place” reversal of the elements using three pointers; no element is physically moved during the reversal, only pointers are changed.

You should try out Program 4.13 on at least three sample chains, the empty chain and chains of length 1 and 2, to convince yourself that you understand the mechanism. For a chain with $n \geq 1$ nodes, the while loop is executed n times and each execution of this loop takes $O(1)$ time. So, the computing time is linear or $O(n)$.

```

template <class T>
void Chain<T>::Reverse ()
{
    // A chain is reversed so that (a1, ..., an) becomes (an, ..., a1).
    ChainNode<T> *current = first,
                  *previous = 0; // previous trails current
    while (current) {
        ChainNode<T> *r = previous;
        previous = current;      // r trails previous
        current = current->link;  // current moves to next node
        previous->link = r;      // link previous to preceding node
    }
    first = previous;
}

```

Program 4.13: Reversing a list

4.3.4 Reusing a Class

Having designed and developed a reusable class for chains, we should now try to reuse this class in applications that require chains. In Section 4.7, we illustrate how polynomials can be implemented by reusing the chain class.

Having said that, however, there are some scenarios where one should not attempt to reuse a class, but rather design a new one. We present two such scenarios below:

- (1) One of the disadvantages of reusing a class (say C_1) to implement another class (say C_2) is that, sometimes, this is less efficient than directly implementing class C_2 . If efficiency is of paramount importance, the latter approach is usually the better course of action. In Section 4.6, we use chains to implement stacks and queues. Since each is an important data structure, which will itself be reused in other applications, we implement them directly to make each as efficient as possible.
- (2) Another scenario in which classes should not be reused is when the operations required by the application are complex and specialized, and therefore not offered by the class. In this situation, also, it is preferable to develop the application directly rather than to “force” the application to reuse an existing class. In Section 4.8, we use chains to solve the problem of finding equivalence classes. The chain operations required by our algorithm are specialized and not likely to be implemented in a reusable chain class. Hence, we implement equivalence classes directly.

EXERCISES

The following exercises assume that chains are defined as in Program 4.6. All functions are to be implemented by employing *ChainIterator* to traverse a chain.

1. Write a C++ template function to output all elements of a chain by overloading the output operator `<<`. Assume that the operator `<<` has been overloaded for the data type *T*.
2. Write a C++ function to compute the minimum of all elements of a chain of integers.
3. Write a C++ function to copy the elements of a chain into an array. Use the STL function *copy* together with array and chain iterators.
4. Let x_1, x_2, \dots, x_n be the elements of a chain. Each x_i is an integer. Write a C++ function to compute the expression $\sum_{i=1}^{n-1} (x_i * x_{i+1})$. [Hint: use two iterators to simultaneously traverse the chain.]
5. Fully code and test the C++ template class *Chain* `<T>`. You must include a constructor, which constructs an initially empty chain; a destructor, which deletes all nodes in the chain; functions to insert at either end of the chain; functions *Front* and *Back* that, respectively, return the first and last elements of the list; a function *Get(i)* that returns the *i*th element in the list; functions to delete from either end; functions to insert as the *i*th element and to delete the *i*th element; and a forward iterator.

4.4 CIRCULAR LISTS

A *circular list* (or, more precisely, a singly-linked circular list) may be obtained by modifying a chain so that the *link* field of the last node points to the first node. Figure 4.13 shows how the chain of Figure 4.2 is modified to form a circular list.

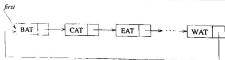


Figure 4.13: A circular list

The circular list structure has some advantages over the chain (as we shall see later). The C++ implementation of circular lists is very similar to that for chains except for some minor differences: To check whether a pointer *current* points to the last node of a circular list, we check for $(current \rightarrow link == first)$ instead of $(current \rightarrow link == 0)$. Secondly, the functions for insertion into and deletion from a circular list must ensure that the *link* field of the last node points to the first node of the list upon completion.

Let us take a look at an operation on a circular list. Suppose we want to insert a new node at the front of the list of Figure 4.14. We have to change the *link* of the node containing x_3 , which requires that we move down the entire length of the list until we find the last node. It is more convenient if the access pointer of a circular list points to the last node rather than to the first (Figure 4.15).



Figure 4.14: Example of a circular list

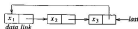


Figure 4.15: Pointing to the last node of a circular list

Now we can write functions that insert at the front (Program 4.14) or at the back of a circular list in $O(1)$ time. To insert e at the back, one only needs to add the statement $last = newNode$ to the else clause of *InsertFront*. The code of Program 4.14 assumes the existence of the template class *CircularList* similar to the class *Chain* of Program 4.6. It is assumed that *CircularList* contains the private data member *last* that points to the last node of the list and that *CircularList* is a friend of *ChainNode*. In some applications, using the structure of Figure 4.13

```

template <class T>
void CircularList<T>::InsertFront(const T& e)
// Insert the element e at the "front" of the circular
// list *this, where last points to the last node in the list.
    ChainNode<T> *newNode = new ChainNode<T>(e);
    if (last) { // nonempty list
        newNode->link = last->link;
        last->link = newNode;
    }
    else { // empty list
        last = newNode;
        newNode->link = newNode;
    }
}

```

Program 4.14: Inserting at the front of a circular list

causes problems as the empty list has to be handled as a special case. To avoid such special cases, we introduce a dummy node, called the *header node*, into each circular list (i.e., each instance of a circular list will contain one additional node). Thus, the empty list will have the representation of Figure 4.16(a) and the list of Figure 4.13 will have the representation of Figure 4.16(b).

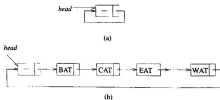


Figure 4.16: Circular lists with a header node

EXERCISES

1. Do Exercise 1 of Section 4.2 for the case of circularly linked lists.
2. Do Exercise 2 of Section 4.2 for the case of circularly linked lists.
3. Do Exercise 3 of Section 4.2 for the case of circularly linked lists.
4. Do Exercise 4 of Section 4.2 for the case of circularly linked lists.
5. Do Exercise 5 of Section 4.2 for the case of circularly linked lists.
6. Do Exercise 6 of Section 4.2 for the case of circularly linked lists.
7. Do Exercise 1 of Section 4.3 for the case of circularly linked lists.
8. Do Exercise 2 of Section 4.3 for the case of circularly linked lists.
9. Do Exercise 3 of Section 4.3 for the case of circularly linked lists.
10. Do Exercise 4 of Section 4.3 for the case of circularly linked lists.
11. Repeat the previous exercise for circularly linked lists with a header node. This time, assume that *CircularListWithHeader* contains a private data member, *header*, that points to the header node.

4.5 AVAILABLE SPACE LISTS

The destructors for chains and circular lists take time linear in the length of the chain or linear list as they delete nodes one at a time. The run time of these destructors may be reduced to $O(1)$ if we maintain our own chain of free (or deleted) nodes. When a new node is needed, we may examine our chain of free nodes. If this chain is not empty, then one of the nodes on it may be made available for use. Only when this chain is empty do we need to invoke *new* to create a new node.

Consider the class *CircularList* of Section 4.4. Let *av* be a static class member of *CircularList*<*T*> of type *ChainNode*<*T*>* that points to the first node in our chain of nodes that have been "deleted." (We implement the chain of deleted nodes directly rather than reusing our class *Chain*<*T*> for efficiency reasons.) Our chain of deleted nodes will henceforth be called the *available-space list* or *av list*. Initially, *av* = 0. Instead of using the functions *new* and *delete*, we shall now use the functions *CircularList*<*T*>::*GetNode* (Program 4.15) and *CircularList*<*T*>::*RetNode* (Program 4.16).

As illustrated by function *CircularList*<*T*>::*CircularList* (Program 4.17), a circular list may now be deleted in a fixed amount of time independent of the number of nodes on the list. Figure 4.17 is a schematic showing the link changes involved in deleting a circular list.

A chain may be deleted in $O(1)$ time if we know its first and last nodes. The instructions:

```

template <class T>
ChainNode <T>* CircularList <T>::GetNode()
// Provide a node for use.
    ChainNode <T>* x;
    if (av) {x = av; av = av->link;}
    else x = new ChainNode <T>;
    return x;
}

```

Program 4.15: Getting a node

```

template <class T>
void CircularList <T>::RetNode(ChainNode <T>* &x)
// Free the node pointed to by x.
    x->link = av;
    av = x;
    x = 0;
}

```

Program 4.16: Returning a node

```

template <class KeyT>
void CircularList <T>::CircularList ()
// Delete the circular list.
    if (last) {
        ChainNode <T>* first = last->link;
        last->link = av; // last node linked to av
        av = first;     // first node of list becomes front of av list
        last = 0;
    }
}

```

Program 4.17: Deleting a circular list

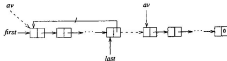


Figure 4.17: Dashed arrows indicate changes involved in deleting a circular list

```
last->link = av;
av = first;
```

accomplish this. When we have our own available space list, we may do other tasks quickly as well. For example, we can generalize the *GetNode* function so that *GetNode(n)* returns a chain or circular list with *n* nodes. Since the available space list already has most of the links correctly set, we can reduce the time spent setting links when creating chains and circular lists of known length.

4.6 LINKED STACKS AND QUEUES

We have already seen how to represent stacks and queues sequentially. In this section we examine how to use linked lists to represent stacks and queues. Figure 4.18 shows a linked stack and a linked queue.

Notice that the direction of links for both the stack and the queue is such that it facilitates insertion and deletion of nodes. In the case of Figure 4.18(a), you can easily add a node at the top or delete one from the top. In Figure 4.18(b), you can easily add a node at the rear, and both addition and deletion can be performed at the front, although for a queue we do not want to add nodes at the front. The public members of the linked stack and queue classes *LinkedStack* and *LinkedQueue* are, respectively, the same as those given in the stack and queue ADTs, ADT 3.1 and ADT 3.2. *LinkedStack* has the single private data member *top*, which points to the top node of the stack and *LinkedQueue* has two private data members *front* and *rear*, which, respectively, point to the first and last nodes of the queue. Both classes use nodes of type *ChainNode<T>* (Section 4.3) and it is assumed that both classes are declared as friends of

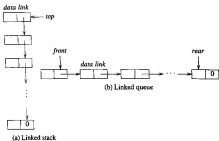


Figure 4.18: Linked stack and queue

ChainNode <T>. The constructor for *LinkedStack* sets *top* to 0 (NULL) and that for *LinkedQueue* sets both *front* and *rear* to 0. The codes for the ADT functions *IsEmpty*, *Top*, *Front* and *Rear* are quite similar to those for the case when an array was used to represent a stack and a queue. Functions for insertion and deletion from linked stacks and queues are presented in Programs 4.19 - 4.22.

```
template <class T>
void LinkedStack <T>::Push(const T& e) {
    top = new ChainNode <T>(e, top);
}
```

Program 4.19: Adding to a linked stack

```

template <class T>
void LinkedStack<T>::Pop()
{ // Delete top node from the stack.
  if (!IsEmpty()) throw "Stack is empty. Cannot delete.";
  ChainNode<T> *delNode = top;
  top = top->link; // remove top node
  delete delNode; // free the node
}

```

Program 4.20: Deleting from a linked stack

```

template <class T>
void LinkedQueue<T>::Push(const T& x)
{
  if (!IsEmpty()) front = rear = new ChainNode(x, 0); // empty queue
  else rear = rear->link = new ChainNode(x, 0); // attach node and update rear
}

```

Program 4.21: Adding to a linked queue

```

template <class T>
void LinkedQueue<T>::Pop()
{ // Delete first element in queue.
  if (!IsEmpty()) throw "Queue is empty. Cannot delete.";
  ChainNode<T> *delNode = front;
  front = front->link; // remove first node from chain
  delete delNode; // free the node
}

```

Program 4.22: Deleting from a linked queue

EXERCISES

1. Develop and test a complete C++ template class for linked stacks.
2. Develop and test a complete C++ template class for linked queues.
3. Consider the hypothetical data object X_2 . X_2 is a linear list with the restriction that although insertions to the list may be made at either end, deletions can be made from one end only. Design a linked-list representation

202 Linked Lists

for *X2*. Develop and test a complete C++ template class implementation for *X2*.

4. Develop and test a C++ template class that implements the *queue* ADT using a circularly linked list.
5. Implement the stack data structure as a derived class of the class *List*<*T*> of Section 4.3. Test your code.
6. Implement the queue data structure as a derived class of the class *List*<*T*> of Section 4.3. Test your code.

4.7 POLYNOMIALS

4.7.1 Polynomial Representation

Let us tackle a reasonable-sized problem using linked lists. This problem, the manipulation of symbolic polynomials, has become a classic example of the use of list processing. In general, we want to represent the polynomial $\alpha(x) = a_n x^{e_n} + \cdots + a_1 x^{e_1}$, where the a_i are nonzero coefficients and the exponents e_i are nonnegative integers such that $e_n > e_{n-1} > \cdots > e_2 > e_1 \geq 0$. We shall define a *Polynomial* class to implement polynomials. Since a polynomial is to be represented by a list, we have *Polynomial* IS-IMPLEMENTED-BY *List*.

Definition: We say that a data object of Type *A* IS-IMPLEMENTED-IN-TERMS-OF a data object of Type *B* if the Type *B* object is central to the implementation of the Type *A* object. This relationship is usually expressed by declaring the Type *B* object as a data member of the Type *A* object. \square

Thus, we shall make the chain object *poly* a data member of *Polynomial*. For this, we employ the chain template class of Program 4.6. Each *ChainNode* will represent a term in the polynomial. To do this, the list template *T* is instantiated to struct *Term*; where, *Term* consists of two data members *coef* and *exp*. We use struct rather than class to define *Term* to emphasize that the data members of *Term* are public. The data members of *Term* are made public so that any function that has access to a *Term* object also has access to its data members. This does not violate data encapsulation for *Polynomial* because the linked list and its contents, including all *Term* objects are all private and cannot be accessed. Assuming that the coefficients are integers, the required class declarations are developed in Program 4.23.

Note that *ChainNodes* contain two fields *data* and *link*, and, since *data* is of type *Term*, *data* contains two fields *coef* and *exp*. For clarity, we draw the nodes in a polynomial as below.

```

struct Term
// All members of Term are public by default.
    int coef;    // coefficient
    int exp;     // exponent
    Term Set(int c, int e) {coef = c; exp = e; return *this;}
};

class Polynomial {
public:
    // public functions defined here
private:
    Chain<Term> poly;
};

```

Program 4.23: Polynomial class definition



Figure 4.19 shows the representation for the polynomials $a = 3x^{14} + 2x^8 + 1$ and $b = 8x^{14} - 3x^{10} + 10x^6$.



Figure 4.19: Representation of $3x^{14} + 2x^8 + 1$ and $8x^{14} - 3x^{10} + 10x^6$

4.7.2 Adding Polynomials

To add two polynomials a and b , we use the list iterators ai and bi to examine these terms starting at the nodes pointed to by $a.first$ and $b.first$. If the exponents of two terms are equal, then the coefficients are added. A new term is created for the result if the sum of the coefficients is not zero. If the exponent of the current term in a is less than the exponent of the current term in b , then a copy of the term in b is created and attached to the *Polynomial* object c . The iterator bi is advanced to the next term in b . Similar action is taken on a if $ai \rightarrow exp > bi \rightarrow exp$. Figure 4.20 illustrates this addition process on the polynomials a and b of the example of Figure 4.19.

Each time a new node is created, its *coef* and *exp* data members are set and the node is appended to the end of c by function *insertBack* (Program 4.11). The complete addition code is specified by the function *operator+*() (Program 4.24).

This is our first complete example of the use of list processing, so it should be carefully studied. The basic algorithm is straightforward, using a merging process that streams along the two polynomials, either copying terms or adding them to the result. Thus, the main while loop of lines 7-21 has three cases depending upon whether the exponents of the terms are ==, <, or >.

Analysis of *operator+*(): The following tasks contribute to the computing time:

- (1) coefficient additions
- (2) exponent comparisons
- (3) inserting a term at the end of a chain.

Let us assume that each of these three tasks, if done once, takes a single unit of time. The total time taken by *operator+*() is then determined by the number of times these tasks are performed. This number clearly depends on how many terms are present in the polynomials a ($+this$) and b . Assume that a and b have m and n terms, respectively:

$$a(x) = a_mx^{e_m} + \cdots + a_1x^{e_1}, b(x) = b_nx^{f_n} + \cdots + b_1x^{f_1}$$

where

$$a_i, b_i \neq 0 \text{ and } e_m > \cdots > e_1 \geq 0, f_n > \cdots > f_1 \geq 0$$

Then clearly the number of coefficient additions can vary as

$$0 \leq \text{coefficient additions} \leq \min\{m, n\}$$

The lower bound is achieved when none of the exponents are equal; the upper

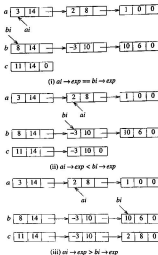


Figure 4.20: Generating the first three terms of $c = a * b$

bound is achieved when the exponents of one polynomial are a subset of the exponents of the other polynomial.

As for exponent comparisons, one comparison is made on each iteration of the first **while** loop. On each iteration either ai or bi or both move to the next

```

1 Polynomial Polynomial::operator+(const Polynomial& b) const
2 { // Polynomials *this (a) and b are added and the sum returned.
3   Term temp;
4   Chain <Term>::ChainIterator ai = poly.begin(),
5                                     bi = b.poly.begin();
6   Polynomial c;
7   while (ai && bi) { // current nodes are not null
8     if (ai->exp == bi->exp) {
9       int sum = ai->coef + bi->coef;
10      if (sum) c.poly.InsertBack(temp.Set(sum, ai->exp));
11      ai++; bi++; // advance to next term
12    }
13    else if (ai->exp < bi->exp) {
14      c.poly.InsertBack(temp.Set(bi->coef, bi->exp));
15      bi++; // next term of b
16    }
17    else {
18      c.poly.InsertBack(temp.Set(ai->coef, ai->exp));
19      ai++; // next term of a
20    }
21  }
22  while (ai) { // copy rest of a
23    c.poly.InsertBack(temp.Set(ai->coef, ai->exp));
24    ai++;
25  }
26  while (bi) { // copy rest of b
27    c.poly.InsertBack(temp.Set(bi->coef, bi->exp));
28    bi++;
29  }
30  return c;
31 }

```

Program 4.24: Adding two polynomials

term in their respective polynomials. Since the total number of terms is $m + n$, the number of iterations and hence the number of exponent comparisons is bounded by $m + n$. You can easily construct a case when $m + n - 1$ comparisons will be necessary — e.g., $m = n$ and

$$e_n > f_n > e_{n-1} > f_{n-1} > \cdots > e_2 > f_2 > e_1 > f_1$$

The maximum number of terms in c is $m + n$, so no more than $m + n$ nonzero terms are inserted at the end of c . In summary, the maximum number of executions of any of the statements in `operator+()` is bounded above by $m + n$. Therefore, the computing time is $O(m + n)$. Since any algorithm that adds two polynomials must look at each nonzero term at least once, our code for `operator+()` is optimal to within a constant factor. \square

4.7.3 Circular List Representation of Polynomials

Figure 4.21 shows the circular list representation of a polynomial. The structure of Figure 4.21, however, causes some problems during addition and other polynomial operations, as the zero polynomial has to be handled as a special case. To avoid such special cases we introduce a header node into each polynomial (i.e., each polynomial, zero or nonzero, will contain one additional node). The `exp` and `coef` data members of this node will not be relevant. Thus, the zero polynomial will have the representation of Figure 4.22(a), and $a = 3x^{14} + 2x^8 + 1$ will have the representation of Figure 4.22(b).

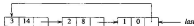


Figure 4.21: Circular list representation of $3x^{14} + 2x^8 + 1$

Using this circular list with header representation, the addition algorithm takes the form given in Program 4.25. This algorithm assumes that the `exp` field of the header node of a polynomial is -1 and that the `begin()` function for `CircularListWithHeader` returns an iterator that points to the node `head` \rightarrow `list`. Now when all terms of a ($*this$) have been examined, `ai` is at the header and `ai` \rightarrow `exp` $= -1$. Since $-1 \leq bi \rightarrow `exp`, the remaining terms of b can be copied by further executions of the while loop. The same is true if all terms of b are examined before those of a . This implies that there is no need for additional code to copy the remaining terms as in Program 4.24.$

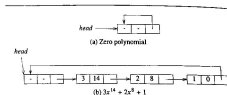


Figure 4.22: Example polynomials

EXERCISES

1. When a class object is passed by value, it is copied into the function's local store. The algorithm for copying the object is specified by the *copy constructor*. If the object's class definition does not define the copy constructor, the default copy constructor is used. The default copy constructor only copies the data members of the object. Define a copy constructor

`Polynomial(const Polynomial& p)`

that copies all the terms of the polynomial *p* into **this*. Do not forget to delete all terms that may be in **this* prior to the invocation of the copy constructor.

2. Overload the C++ input operator `<<` so that objects of type `Polynomial` may be input using this operator. Your code should read in *n* pairs of coefficients and exponents, (c_i, e_i) , $1 \leq i \leq n$, of a univariate polynomial and convert the polynomial into the circularly linked list representation described in this section. Assume that $e_i > e_{i+1}$, $1 \leq i < n$, and that $c_i \neq 0$, $1 \leq i \leq n$. Show that this operation can be performed in time $O(n)$.
3. Let *a* and *b* be two polynomials represented as circular lists with header nodes. Write a C++ function to compute the product polynomial $c = a * b$. Your code should leave *a* and *b* unaltered. Show that if *n* and *m* are the number of terms in *a* and *b*, respectively, then this multiplication can be carried out in time $O(nm^2)$ or $O(mn^2)$. If *a* and *b* are dense, show that the multiplication takes $O(nm)$.

```

1 Polynomial Polynomial::operator+(const Polynomial& b) const
2 { // Polynomials *this (a) and b are added and the sum returned.
3     Term temp;
4     CircularListWithHeader<Term>::iterator ai = poly.begin(),
5                                         bi = b.poly.begin();
6     Polynomial c; // assume constructor sets head->exp = -1
7     while (1) {
8         if (ai->exp == bi->exp) {
9             if (ai->exp == -1) return c;
10            int sum = ai->coef + bi->coef;
11            if (sum) c.poly.InsertBack(temp.Set(sum, ai->exp));
12            ai++; bi++; // advance to next term
13        }
14        else if (ai->exp < bi->exp) {
15            c.poly.InsertBack(temp.Set(bi->coef, bi->exp));
16            bi++; // next term of b
17        }
18        else {
19            c.poly.InsertBack(temp.Set(ai->coef, ai->exp));
20            ai++; // next term of a
21        }
22    }
23 }

```

Program 4.25: Adding two polynomials represented as circular lists with header nodes

4. Write a C++ function to evaluate a polynomial at the point x , where x is a real number. Assume that the polynomial is represented as a circularly linked list with a header node.
5. [Programming Project] Develop a C++ class *Polynomial* to represent and manipulate univariate polynomials with integer coefficients (use circular linked lists with header nodes). Each term of the polynomial will be represented as a node. Thus, a node in this system will have three data members as below:

coef	exp	link
------	-----	------

Each polynomial is to be represented as a circular list with header node. To delete polynomials efficiently, we need to use an available-space list and associated functions as described in Section 4.3. The external (i.e., for input or output) representation of a univariate polynomial will be assumed to be a sequence of integers of the form: $n, e_1, a_1, e_2, a_2, e_3, a_3, \dots, e_k, a_k$, where e_i represents an exponent and a_i a coefficient; n gives the number of terms in the polynomial. The exponents are in decreasing order— $e_1 > e_2 > \dots > e_k$.

Write and test the following functions:

- `istream& operator>>(istream& is, Polynomial& x)`: Read in an input polynomial and convert it to its circular list representation using a header node.
- `ostream& operator<<(ostream& os, Polynomial& x)`: Convert x from its linked list representation to its external representation and output it.
- `Polynomial::Polynomial(const Polynomial& a)` [Copy Constructor]: Initialize the polynomial `*this` to the polynomial a .
- `const Polynomial& Polynomial::operator=(const Polynomial& a)` `const` [Assignment Operator]: Assign polynomial a to `*this`.
- `Polynomial::~Polynomial()` [Destructor]: Return all nodes of the polynomial `*this` to the available-space list.
- `Polynomial operator+ (const Polynomial& b) const` [Addition]: Create and return the polynomial `*this + b`.
- `Polynomial operator- (const Polynomial& b) const` [Subtraction]: Create and return the polynomial `*this - b`.
- `Polynomial operator* (const Polynomial& b) const` [Multiplication]: Create and return the polynomial `*this * b`.
- `float Polynomial::Evaluate(float x) const`: Evaluate the polynomial `*this` at x and return the result.

4.8 EQUIVALENCE CLASSES

Let us put together some of the ideas discussed so far on linked and sequential representations to solve a problem that arises in the design and manufacture of very large-scale integrated (VLSI) circuits. One of the steps in the manufacture of a VLSI circuit involves exposing a silicon wafer using a series of masks. Each

mask consists of several polygons. Polygons that overlap are electrically equivalent. Electrical equivalence specifies a relationship among mask polygons. This relation has several properties that it shares with other relations, such as the conventional mathematical equivalence. Suppose we denote an arbitrary relation by the symbol \equiv , and suppose that

- (1) For any polygon x , $x \equiv x$ (e.g., x is electrically equivalent to itself). Thus, \equiv is *reflexive*.
- (2) For any two polygons x and y , if $x \equiv y$, then $y \equiv x$. Thus, the relation \equiv is *symmetric*.
- (3) For any three polygons x , y , and z , if $x \equiv y$ and $y \equiv z$, then $x \equiv z$ (e.g., if x and y are electrically equivalent and y and z are also, then so also are x and z). The relation \equiv is *transitive*.

Definition: A relation \equiv over a set S , is said to be an *equivalence relation* over S iff it is symmetric, reflexive, and transitive over S . \square

Equivalence relations are numerous. For example, the "equal to" ($=$) relationship is an equivalence relation, since (1) $x = x$, (2) $x = y$ implies $y = x$, and (3) $x = y$ and $y = z$ implies $x = z$. One effect of an equivalence relation is to partition the set S into equivalence classes such that two members x and y of S are in the same equivalence class iff $x \equiv y$. For example, if we have 12 polygons numbered 0 through 11 and the following overlap pairs are defined:

0 = 4, 3 = 1, 6 = 10, 8 = 9, 7 = 4, 6 = 8, 3 = 5, 2 = 11, and 11 = 0

then, as a result of the reflexivity, symmetry, and transitivity of the relation \equiv , we get the following partitioning of the 12 polygons into three equivalence classes:

{0, 2, 4, 7, 11}; {1, 3, 5}; {6, 8, 9, 10}

These equivalence classes are important, as each equivalence class defines a *signal net*. The signal nets can be used to verify the correctness of the masks.

Our algorithm to determine equivalence classes works in essentially two phases. In the first phase the equivalence pairs (i, j) are read in and stored. In phase two we begin at 0 and find all pairs of the form $(0, j)$. The values 0 and j are in the same class. By transitivity, all pairs of the form (j, k) imply k is in the same class as 0. We continue in this way until the entire equivalence class containing 0 has been found and output. Then we find an object not yet output. This is in a new equivalence class. The objects in this equivalence class are found as before and output.

The first design for our equivalence class algorithm might be as in Program 4.26. Let m and n represent the number of input pairs and the number of objects, respectively. We need to select a data structure to hold these pairs. Easy random access would suggest a Boolean array, say $\text{pairs}[m][n]$. The element $\text{pairs}[i][j] =$

```

void Equivalence()
{
    initialize;
    while more pairs
    {
        input the next pair (i,j);
        process this pair;
    }
    initialize for output;
    for (each object not yet output)
        output the equivalence class that contains this object;
}

```

Program 4.26: First version of equivalence algorithm.

true if and only if (i,j) is an input pair. However, using such an array could potentially be quite wasteful of space since very few of the array elements may be used. Any algorithm that uses this data structure would require $\Theta(n^2)$ time, just to initialize the array.

These considerations lead us to consider a chain to represent each row of the aforementioned array pairs. Each node on a chain requires only a *data* and a *link* field. However, we still need random access to the i th row. For this, we use a one-dimensional array, *first*[n], with *first*[i] a pointer to the first node in the chain for row i .

Looking at the second phase of the algorithm we need a mechanism that tells us whether or not object i is yet to be printed. A Boolean array, *out*[n], can be used for this. Program 4.27 gives the next refinement of our equivalence algorithm.

Let us simulate the algorithm, as we have it so far, on the input data set

$0 = 4$, $3 = 1$, $6 = 10$, $8 = 9$, $7 = 4$, $6 = 8$, $3 = 5$, $2 = 11$, and $11 = 0$

After the while loop is completed the chains look as in Figure 4.23. For each relation $i = j$, two nodes are used. *first*[i] points to a chain of nodes that contains every number directly equivalent to i by an input relation.

In phase two, we scan the *first* array in the order i , $0 \leq i < n$. For each i such that *out*[i] is false, the elements in the list *first*[i] are output. To enable the processing of the remaining elements which, by transitivity, belong in the same class as i , a linked stack is created. Thus, a node may initially be on one of the chains given by *first*[] and later be on a linked stack of elements waiting to be processed. The complete function is given in Program 4.28.

```

void Equivalence()
{
    read n;           // read in number of objects
    initialize first[0:n-1] to 0 and out[0:n-1] to false;
    while more pairs // input pairs
    {
        read the next pair (i,j);
        put j on the chain first[i];
        put i on the chain first[j];
    }
    for (i = 0; i < n; i++)
        if (out[i]) {
            out[i] = true;
            output the equivalence class that contains object i;
        }
}

```

Program 4.27: A more detailed version of equivalence algorithm



Figure 4.23: Lists after pairs have been input

234 Linked Lists

```

class ENode {
friend void Equivalence ();
public:
    ENode(int d = 0) // constructor
        {data = d; link = 0;}
private:
    int data;
    ENode *link;
};

void Equivalence ()
// Input the equivalence pairs and output the equivalence classes.
{ifstream inFile("equiv.in", ios::in); // "equiv.in" is the input file
if (!inFile) throw "Cannot open input file.";
int i, j, n;
inFile >> n; // read number of objects
// initialize first and out
ENode *first = new ENode[n];
bool *out = new bool[n];
// use STL function fill to initialize
fill(first, first + n, 0);
fill(out, out + n, false);

// Phase 1: input equivalence pairs
inFile >> i >> j;
while (inFile.good()) { // check end of file
    first[i] = new ENode(j, first[j]);
    first[j] = new ENode(i, first[i]);
    inFile >> i >> j;
}

// Phase 2: output equivalence classes
for (i = 0; i < n; i++)
    if (!out[i]) { // i needs to be output
        cout << endl << "A new class: " << i;
        out[i] = true;
        ENode *x = first[i]; ENode *top = 0; // initialize stack
        while (1) { // find rest of class
            while (x) { // process the list
                j = x->data;
                if (!out[j]) {

```

```

    cout << ", " << j;
    out[j] = true;
    ENode *y = x->link;
    x->link = top;
    top = x;
    x = y;
}
else x = x->link;
} // end of while(x)
if (!top) break;
x = first[top->data];
top = top->link; // unstack
} // end of while(1)
} // end of if (!out[i])
for (i = 0; i < n; i++)
    while (first[i]) {
        ENode *delnode = first[i];
        first[i] = delnode->link;
        delete delnode;
    }
delete [] first; delete [] out;
}

```

Program 4.28: C++ function to find equivalence classes

Analysis of Equivalence: The initialization of *first* and *out* takes $O(n)$ time. The processing of each input pair in phase 1 takes a constant amount of time. Hence, the total time for this phase is $O(n + m)$, where m is the number of input pairs. In phase 2 each node is put onto the linked stack at most once. Since there are only $2m$ nodes and the *for* loop is executed n times, the time for this phase is $O(m + n)$. Hence, the overall computing time is $O(m + n)$. Any algorithm that processes equivalence relations must look at all m equivalence pairs and at all n objects at least once. Thus, no algorithm can have a computing time less than $\Theta(m + n)$. This means that function *Equivalence* is optimal to within a constant factor. Unfortunately, the space required by the algorithm is also $O(m + n)$. In Chapter 5 we shall see an $O(n)$ -space solution to this problem.

EXERCISES

1. Rewrite function *Equivalence* (Program 4.28) using an array of objects of type `Choice<int>` rather than an array of type `ENode*`. Also, use an object of type `Stack<int>` (see Section 3.2) rather than the custom linked stack used in Program 4.28. What can you say about the expected relative

performance of your function and Program 4.28? Which requires more space?

2. Rewrite function *Equivalence* (Program 4.28) using dynamic arrays and array doubling in place of chains (see Sections 2.5 and 3.1.1). Compare the worst-case space and time complexities of the two versions of *Equivalence*.

4.9 SPARSE MATRICES

4.9.1 Sparse Matrix Representation

In Chapter 2, we saw that when matrices were sparse (i.e., many of the entries were zero), then much space and computing time could be saved if only the nonzero terms were retained explicitly. In the case where these nonzero terms did not form any “nice” pattern such as a triangle or a band, we devised a sequential scheme in which each nonzero element was represented by a structure with three data members: row, column, and value. The sequential representation of Chapter 2 permits easy access of matrix terms by row. However, accessing all the terms in a specific column of a matrix is difficult. To provide easy access both by row and by column, we devise a linked representation for a sparse matrix. In the data representation we use, each nonzero element is in two circular lists; one is a row list and the other a column list. So, we have a circular list for each row and each column of the matrix. Each circular list has a header node.

Our sparse matrix representation uses nodes of the type *MatrixNode*. This class has a Boolean field *head*, which is used to distinguish between header nodes and nodes that represent nonzero elements. Each header node has three additional fields: *down*, *right*, and *next*. The total number of header nodes is $\max(\text{number of rows, number of columns})$. The header node for row i is also the header node for column i . The *down* field of a header node is used to link into a column list; the *right* field is used to link into a row list. The *next* field links the header nodes together.

All other nodes have six fields: *head*, *row*, *col*, *value*, *down*, and *right* (Figure 4.24). The *down* field is used to link to the next nonzero term in the same column and the *right* field links to the next nonzero term in the same row. Thus, if $a_{ij} \neq 0$, then there is a node with *head* = *false*, *value* = a_{ij} , *row* = i , and *col* = j . This node is linked into the circular linked lists for row i and column j . Hence, the node is simultaneously in two different lists.

As noted earlier, each header node is in three lists: a row list, a column list, and a list of header nodes. The list of header nodes itself has a header node, *H*, that is identical to the nodes used to represent nonzero elements. The *row* and



Figure 4.24: Node structure for sparse matrices

col fields of this node are used to store the matrix dimensions. The entire matrix is represented by class *MMatrix* with the data member *headnode*, which points to *H*.

$$\begin{pmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{pmatrix}$$

Figure 4.25: 5×4 sparse matrix *a*

The linked structure obtained for the 5×4 matrix, *a*, of Figure 4.25 is shown in Figure 4.26. Although Figure 4.26 does not show the values of the *head* fields, these values are readily determined from the node structure shown. For each nonzero term of *a*, we have one six-field node that is in exactly one column list and one row list. The header nodes are marked H0 to H4 and are drawn twice to simplify the figure. As seen in the figure, the right field of *headnode* is used to link into the list of header nodes. Notice that the whole matrix may be referenced through the header node, *headnode*, of the list of header nodes.

If we wish to represent an $n \times m$ sparse matrix with *r* nonzero terms, the number of nodes needed is $\max\{n, m\} + r + 1$. Although each node may require several words of memory, the total storage needed will be less than nm for sufficiently small *r*.

The required node structure may be defined in C++ using an anonymous

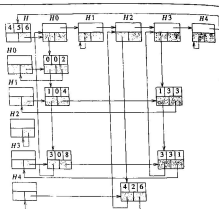


Figure 4.26: Linked representation of the sparse matrix, of Figure 4.25 (the *Head* field of a node is not shown)

union as in Program 4.29. Since all nodes contain fields *head*, *down* and *right*, these are declared outside the anonymous union. Header nodes also contain *next* while all other nodes contain *row*, *col*, and *value*. These fields are all incorporated into the struct *Triple*. Space is allocated by the union declaration for the larger of the fields *next* and *triple* (which is an object of type *Triple*).

```

struct Triple {int row, col, value};
class Matrix; // forward declaration
class MatrixNode {
friend class Matrix;
friend istream& operator>>(istream&, Matrix&); // for reading in a matrix
private:
    MatrixNode *down, *right;
    bool head;
    union { // anonymous union
        MatrixNode *next;
        Triple triple;
    };
    MatrixNode(bool, Triple *); // constructor
};

MatrixNode::MatrixNode(bool b, Triple * t) // constructor
{
    head = b;
    if (b) {right = down = this;} // row/column header node
    else triple = *t; // element node or header node for list of header nodes
}

class Matrix {
friend istream& operator>>(istream&, Matrix&);
public:
    ~Matrix(); // destructor
private:
    MatrixNode *headnode;
};

```

Program 4.29: Class definitions for sparse matrices

4.9.2 Sparse Matrix Input

The first operation we shall consider is that of reading in a sparse matrix and obtaining its linked representation. We assume that the first input line gives us the number of rows, the number of columns, and the number of nonzero terms in the matrix. Each subsequent input line is a triple of the form (i, j, a_{ij}) . These triples consist of the *row*, *col*, and *value* data members of the nonzero terms of the matrix. We also assume that these triples are ordered by rows and within rows by columns.

220 Linked Lists

For example, the input for the 5×4 sparse matrix of Figure 4.25, which has six nonzero terms, would take the following form: 5, 4, 6; 0, 0, 2; 1, 0, 4; 1, 3, 3; 3, 0, 8; 3, 3, 1; 4, 2, 6. The code for the input operator `>>` (Program 4.30) makes use of an auxiliary array `head`, of size `max(s.row, s.col)`. `head[i]`, which is a pointer to the header node for column `i` and hence also for row `i`, enables efficient random access to columns while the input matrix is set up. Function `operator>>` first creates the header nodes (but doesn't link them together) and then inputs the matrix elements one at a time. Each element is added to its row and column lists upon input. The `next` field of header node `i` is used initially to keep track of the last node in column `i`. Eventually, in line 31, the header nodes are linked together through this field.

Analysis of `operator>>`: Assuming that `new` works in $O(1)$ time, all the header nodes may be set up in $O(\max\{n, m\})$ time, where n is the number of rows and m the number of columns in the matrix being input. Each nonzero term is set up in $O(1)$ time because of the use of the variable `last` and a random access scheme for the bottommost node in each column list. Hence, the `for` loop of lines 15-26 takes $O(r)$ time. The rest of the algorithm takes $O(\max\{n, m\})$ time. The total time is therefore $O(\max\{n, m\} + r) = O(n + m + r)$. Note that this time is asymptotically better than the input time of $O(nr)$ for an $n \times m$ matrix using a two-dimensional array but slightly worse than that for the sequential sparse method of Section 2.3. \square

4.9.3 Deleting a Sparse Matrix

All the nodes of a sparse matrix may be returned one at a time using `delete`. A faster way to return the nodes is to set up an available-space list (Section 4.5). Assume that `av` points to the first node of this list and that this list is linked through the field `right`. Function `Matrix::~Matrix()` (Program 4.31) deletes a sparse matrix in an efficient way.

Analysis of `Matrix()`: Since each node is in only one row list, it is sufficient to return all the row lists of the matrix. Each row list is circularly linked through the field `right`. Thus, nodes need not be returned one by one, as a circular list can be deleted in $O(1)$ time. The computing time for Program 4.31 is readily seen to be $O(n + m)$. Note that even if the available-space list had been linked through the field `down`, deleting still could have been carried out in $O(n + m)$ time. \square

```

1 istream& operator>>(istream& is, Matrix& matrix)
2 { // Read in a matrix and set up its linked representation.
3   Triple t;
4   is >> t.row >> t.col >> t.value; // matrix dimensions
5   int p = max(t.row, t.col);
6   // set up header node for list of header nodes.
7   matrix.headnode = new MatrixNode(false, &t);
8   if (p == 0) { matrix.headnode->right = matrix.headnode; return is; }
9   // at least one row or column
10  MatrixNode **head = new MatrixNode * [p];
11  for (int i = 0; i < p; i++)
12    head[i] = new MatrixNode(true, 0);
13  int currentRow = 0;
14  int MatrixNode *last = head[0]; // last node in current row
15  for (i = 0; i < t.value; i++) // input triples
16  {
17    Triple t;
18    is >> t.row >> t.col >> t.value;
19    if (t.row > currentRow) { // close current row
20      last->right = head[currentRow];
21      currentRow = t.row;
22      last = head[currentRow];
23    } // end of if
24    last = last->right = new MatrixNode(false, &t); // link new node into row list
25    head[t.col]->next = head[t.col]->next->down = last; // link into column list
26  } // end of for
27  last->right = head[currentRow]; // close last row
28  for (i = 0; i < t.col; i++) head[i]->next->down = head[i]; // close all column lists
29  // link the header nodes together
30  for (i = 0; i < p - 1; i++) head[i]->next = head[i + 1];
31  head[p - 1]->next = matrix.headnode;
32  matrix.headnode->right = head[0];
33  delete [] head;
34  return is;
35 }

```

Program 4.30: Reading in a sparse matrix

```

Matrix::Matrix()
{
    // Return all nodes to the av list. This list is a chain linked via the right
    // field. av is a static variable that points to the first node of the av list.
    if (!headnode) return; // no nodes to delete
    MatrixNode *x = headnode->right;
    headnode->right = av; av = headnode; // return headnode
    while (x != headnode) { // erase by rows
        MatrixNode *y = x->right;
        x->right = av;
        av = y;
        x = x->left; // next row
    }
    headnode = 0;
}

```

Program 4.31: Deleting a sparse matrix

EXERCISES

In Exercises 1 to 5, the sparse matrix representation described in this section is assumed.

1. Write the C++ function, `operator+(const Matrix& b) const`, which returns the matrix `*this + b`. Show that if `*this` and `b` are $n \times m$ matrices with r_a and r_b nonzero terms, then this addition can be carried out in $O(n + m + r_a + r_b)$ time.
2. Write the C++ function, `operator*(const Matrix& b) const`, which returns the matrix `*this * b`. If `a` is an $n \times m$ matrix with r_a nonzero terms and if `*this` is an $n \times m$ matrix with r_a nonzero terms and `b` is an $m \times p$ matrix with r_b nonzero terms, then this multiplication can be done in $O(pr_a + nr_b)$ time. Can you think of a way to do the multiplication in $O(\min\{pr_a, nr_b\})$ time?
3. Write the C++ function `operator<<()`, which outputs a sparse matrix as triples (i, j, a_{ij}) . The triples are to be output by rows and within rows by columns. Show that this operation can be performed in time $O(n + r_a)$ if there are r_a nonzero terms in the matrix. n is the number of rows in the matrix.
4. Write the C++ function, `Transpose()`, which transposes a sparse matrix. What is the computing time of your function?

5. Write and test a copy constructor for sparse matrices. What is the computing time of your copy constructor?
6. *[Programming Project]* A simpler and more efficient representation for sparse matrices can be obtained when one is limited to the operations of addition, subtraction, and multiplication. Now, nodes have the data members *rowLink*, *colLink*, *row*, *col*, and *value*. Each nonzero term is represented by a node. These nodes are linked together to form two circular lists. The first list, the row list, is made up by linking nodes by rows and within rows by columns. The linking is done via the *right* data member. The second list, the column list, is made up by linking nodes via the *down* data member. In this list, nodes are linked by columns and within columns by rows. These two lists share a common header node. In addition, a node is added to contain the dimensions of the matrix. Draw the resulting representation for the matrix of Figure 4.25.

Write a C++ class for this representation. You must include the following functions. What is the computing time of each of your functions? How do these times compare with the corresponding times for the representation of this section?

- (a) `istream& operator>>(istream& is, Matrix& m);`
Read in the matrix and set it up according to the representation of this section. The first input line gives the matrix dimensions. The next several lines contain one triple, (row, column, value), each. The last triple ends the input file. These triples are in increasing order by rows. Within rows, the triples are in increasing order of columns. The data is to be read in one line at a time and converted to internal representation.
- (b) `ostream& operator<<(ostream& os, const Matrix& m);`
Output the terms of *m*. To do this, you will have to design a suitable output format. The output should be ordered by rows and within rows by columns.
- (c) `Matrix::Matrix(const Matrix& a) [Copy Constructor];`
Initialize the sparse matrix **this* to the sparse matrix *a*.
- (d) `const Matrix& Matrix::operator=(const Matrix& a) const [Assignment Operator];` Assign sparse matrix *a* to **this*.
- (e) `Matrix::~Matrix() [Destructor];`
Return all nodes of the sparse matrix **this* to the available-space list.
- (f) `Matrix Matrix::operator+(const Matrix& b) const;`
Create and return the sparse matrix **this* + *b*.
- (g) `Matrix Matrix::operator-(const Matrix& b) const;`
Create and return the sparse matrix **this* - *b*.

- (b) *Matrix* *Matrix::operator** (*const Matrix& b*) *const*:
Create and return the sparse matrix **this* * *b*.
 - (c) *Matrix* *Matrix::Transpose* () *const*:
Create and return the transpose of **this*.
7. Compare the sparse representations of Exercise 6 and this section with respect to the time needed to output the elements in an arbitrary row or column.
 8. [Programming Project] Implement a complete linked-list system to perform arithmetic on sparse matrices using the representation of this section. You must include all the functions listed in Exercise 6.

4.10 DOUBLY LINKED LISTS

So far we have been working chiefly with chains and singly linked circular lists. For some problems these would be too restrictive. One difficulty with these lists is that if we are pointing to a specific node, say *p*, then we can move only in the direction of the links. The only way to find the node that precedes *p* is to start at the beginning of the list. The same problem arises when one wishes to delete an arbitrary node from a singly linked list. As can be seen from Example 4.4, easy deletion of an arbitrary node requires knowing the preceding node. If we have a problem in which it is necessary to move in either direction or in which we must delete arbitrary nodes, then it is useful to have doubly linked lists. Each node now has two link fields, one linking in the forward direction and the other linking in the backward direction.

A node in a doubly linked list has at least two fields—e.g., *left* (left link), and *right* (right link). Usually, each node will have a *data* field as well. A doubly linked list may or may not be circular and may or may not have a header node. A sample doubly linked circular list with header node is shown in Figure 4.27. This list has a header node plus three additional nodes. As was true in the earlier sections, header nodes are convenient for the algorithms. Now suppose that *p* points to any node in a doubly linked list. So, *p* = *p*→*left*→*right* == *p*→*right*→*left*. This formula reflects the essential virtue of this structure—namely, that one can go back and forth with equal ease. When header nodes are used, an empty list has exactly one node—the header node (Figure 4.28). Program 4.32 contains the class definition of a doubly linked list of integers. To work with these lists we must be able to insert and delete nodes. Function *DBLList::Delete* (Program 4.33) deletes node *x* from the list. Following the deletion, *x* points to a node that is no longer part of the list. Figure 4.29 shows how the function works on a doubly linked list with only a single node. Even though the *right* and *left* data members of node *x* still point to the header node, this node

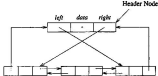


Figure 4.27: Doubly linked circular list with header node



Figure 4.28: Empty doubly linked circular list with header node

has effectively been removed, as there is no way to access x through *first*. Insertion is only slightly more complex (see Program 4.34).

EXERCISES

1. Let x be a node in a singly linked circular list. Write a C++ function to delete the data in this node. Following the deletion, the number of nodes in the list is one less than before the deletion. Your function must run in $O(1)$ time. (Hint: Instead of deleting the node x , copy the data from the node, if any, that is next to x and delete that node instead.)
2. Write a function `void DbfList::Concatenate(DbfList m)` to concatenate the two lists `*this` and `m`. On completion of the function, the resulting list should be stored in `*this` and the list `m` should contain the empty list. Your function must run in $O(1)$ time.

```

class DbList;

class DbListNode {
friend class DbList;
private:
    int data;
    DbListNode *left, *right;
};

class DbList {
public:
    // List manipulation operations
    .
    .
private:
    DbListNode *first; // points to header node
};

```

Program 4.32: Class definition of a doubly linked list

```

void DbList::Delete(DbListNode *x)
{
    if (x == first) throw "Deletion of header node not permitted";
    else {
        x->left->right = x->right;
        x->right->left = x->left;
        delete x;
    }
}

```

Program 4.33: Deletion from a doubly linked circular list

3. Devise a linked representation for a list in which insertions and deletions can be made at either end in $O(1)$ time. Such a structure is called a *deque*. Write functions to insert and delete at either end.

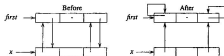


Figure 4.29: Deletion from a doubly linked circular list

```

void DblList::Insert(DblListNode *p, DblListNode *x)
// insert node p to the right of node x
    p->left = x; p->right = x->right;
    x->right->left = p; x->right = p;
}

```

Program 4.34: Insertion into a doubly linked circular list

4. Consider the operation XOR (exclusive OR, also written as \oplus) defined as follows (for i, j binary):

$$i \oplus j = \begin{cases} 0 & \text{if } i \text{ and } j \text{ are identical} \\ 1 & \text{otherwise} \end{cases}$$

This definition differs from the usual OR of logic, which is defined as

$$i \text{ OR } j = \begin{cases} 0 & \text{if } i = j = 0 \\ 1 & \text{otherwise} \end{cases}$$

The definition can be extended to the case in which i and j are binary strings (i.e., take the XOR of corresponding bits of i and j). So, for example, if $i = 10110$ and $j = 01100$, then $i \text{ XOR } j = i \oplus j = 11010$. Note that

$$a \oplus (a \oplus b) = (a \oplus a) \oplus b = b$$

and

$$(a \oplus b) \oplus b = a \oplus (b \oplus b) = a$$

This notation gives us a space-saving device for storing the right and left links of a doubly linked list. The nodes will now have only two data members: *data* and *link*. If *l* is to the left of node *x* and *r* to its right, then $x \rightarrow \text{link} = l \oplus r$ if *x* is the leftmost node of a non-circular list, *l* = 0, and if *x* is the rightmost node, *r* = 0. For a new doubly linked list class in which the link field of each node is the exclusive or of the addresses of the nodes to its left and right, do the following.

- (a) Write a C++ function to traverse the doubly linked list from left to right, printing out the contents of the data field of each node.
 - (b) Write a C++ function to traverse the list from right to left, printing out the contents of the data field of each node.
5. [Programming Project]: Implement a C++ template class for doubly linked circular lists with header nodes. You must include a constructor, copy constructor and destructor as well as functions to insert and delete. A bidirectional iterator must be included as well.

4.11 GENERALIZED LISTS

4.11.1 Representation of Generalized Lists

In Chapter 2 a linear list was defined to be a finite sequence of $n \geq 0$ elements, a_0, \dots, a_{n-1} , which we write as (a_0, \dots, a_{n-1}) . The elements of a linear list are restricted to atoms; thus, the only structural property a linear list has is that of position (i.e., a_i precedes a_{i+1} , $0 \leq i < n-1$). Relaxing this restriction on the elements of a list and permitting them to have a structure of their own leads to the notion of a generalized list. Now, the elements a_i , $0 \leq i \leq n-1$, may be either atoms or lists.

Definition: A generalized list, is a finite sequence of $n \geq 0$ elements, a_0, \dots, a_{n-1} , where a_i is either an atom or a list. The elements a_i , $0 \leq i \leq n-1$, that are not atoms are said to be the *sublists*. \square

Let $A = (a_0, \dots, a_{n-1})$ be a list. *A* is the name of the list (a_0, \dots, a_{n-1}) and *n* is its length. By convention, all list names are represented by capital letters. Lowercase letters are used to represent atoms. If $n \geq 1$, then a_0 is the *head* of *A*, and (a_1, \dots, a_{n-1}) is the *tail* of *A*. Some examples of generalized lists are

- (1) $A = ()$ the null, or empty, list; its length is zero.
- (2) $B = (a, (b, c))$ a list of length two; its first element is the atom *a*, and its

second element is the linear list $\langle b, c \rangle$.

(3) $C = \langle B, B, () \rangle$: a list of length three whose first two elements are the list B , and the third element is the null list.

(4) $D = \langle a, D \rangle$: a recursive list of length two; D corresponds to the infinite list $\langle a, \langle a, \langle a, \dots \rangle \rangle$.

A is the empty list. For list B , we have $\text{head}(B) = 'a'$ and $\text{tail}(B) = \langle \langle b, c \rangle \rangle$; $\text{tail}(B)$ also has a head and tail, which are $\langle b, c \rangle$ and $()$, respectively. Looking at list C , we see that $\text{head}(C) = B$ and $\text{tail}(C) = \langle B, () \rangle$. Continuing, we have $\text{head}(\text{tail}(C)) = B$ and $\text{tail}(\text{tail}(C)) = ()$, both of which are lists.

Two important consequences of our definition for a list are: (1) lists may be shared by other lists as in example (3), where list B makes up two of the sublists of C ; and (2) lists may be recursive as in example (4). The implications of these two consequences for the data structures needed to represent lists will become evident.

First, let us restrict ourselves to the situation in which the lists being represented are neither shared nor recursive. To see where this notion of a list may be useful, consider the problem of representing polynomials in several variables. For example,

$$P(x, y, z) = x^{10}y^3z^1 + 2x^8y^3z^2 + 3x^8y^2z^2 + x^4y^4z + 6x^3y^4z + 2yz$$

Each term of P may be represented using a structure that has four fields: *coef*, *expo*, *expy*, and *expz*. Going this route would require us to define one structure for polynomials in one variable, another for those in two variables, and so on. Alternatively, we could define a structure with a large number of exponent fields and use only as many as needed. Neither of these solutions is elegant.

If we rewrite $P(x, y, z)$ as

$$((x^{10} + 2x^8)y^3 + 3x^8y^2)z^2 + ((x^4 + 6x^3)y^4 + 2y)z$$

we see that $P(x, y, z)$ may be viewed as a generalized list. Every polynomial can be written in this fashion, factoring out a main variable z , followed by a second variable y , and so on. Looking carefully now at $P(x, y, z)$, we see that there are two terms in the variable z , Cz^2 and Dz , where C and D are polynomials themselves, but in the variables x and y . Looking more closely at $C(x, y)$, we see that it is of the form $Ey^3 + Fy^2$, where E and F are polynomials in x . Continuing in this way, we see that every polynomial consists of a variable plus coefficient-exponent pairs. Each coefficient is itself a polynomial (in one less variable) if we regard a single numerical coefficient as a polynomial in zero variables.

We see that every polynomial, regardless of the number of variables in it, can be represented using nodes of the type *Polynode*, defined as

230 Linked Lists

```
enum Triple {var, pr, no};
class PolyNode
{
    PolyNode *next;      // link field
    int exp;
    Triple trio;
    union {
        char vble;
        PolyNode *down;  // link field
        int coef;
    };
};
```

In this representation, there are three types of nodes, depending on the value of *trio*. If *trio* == *var*, then the node is the header node for a list; in this case, the field *vble* is used to indicate the name of the variable on which that list is based and the *exp* field is set to 0. (Note that the type of the data member *vble* can be changed to list if all variables are kept in a table and *vble* just gives the corresponding table index.) If *trio* == *pr*, then the coefficient is itself a list and is pointed by the field *down*. If *trio* == *no*, then the coefficient is an integer and is stored in the field *coef*. In the last two cases, *exp* represents the exponent of the variable on which that list is based.

Figure 4.30 gives the representation of the polynomial $3x^2y$. Here *first* points to a header node, which indicates that the upper list is based on *y*. Hence the next term in the list refers to the term $y^1 = y$. The coefficient of this term is the polynomial represented by the lower list. This polynomial is $3x^2$. Figure 4.31 gives the representation of the polynomial $P(x, y, z)$ defined earlier. For simplicity, the *trio* field is omitted from Figure 4.31. The value of this field for each node is self-evident.

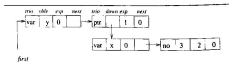


Figure 4.30: Representation of $3x^2y$

Every generalized list can be represented using the node structure



Figure 4.31: $((x^{10} + 2x^8)y^3 + 3x^6y^3)x^2 + ((x^8 + 6x^3)y^4 + 2y)x$

<code>isg = false/true</code>	<code>data/down</code>	<code>next</code>
-------------------------------	------------------------	-------------------

This data structure may be defined in C++ as:

template <class T> class GenList; // forward declaration

```
template <class T>
class GenListNode {
friend class GenList <T>;
private:
    GenListNode <T> *next;
    bool isg;
    union {
        T data;
        GenListNode <T> *down;
    };
};
```

```
template <class T>
class GenList {
public:
    // List manipulation operations

private:
    GenListNode <T> *first;
};
```


232 Linked Lists

The *data/atom* field holds an atom if *head(A)* is an atom and holds a pointer to the list representation of *head(A)* if *head(A)* is a list. (The exercises examine how a multivariate polynomial may be stored using this structure.) Using this node structure, the example lists (1) to (4) (page 228) have the representation shown in Figure 4.32.

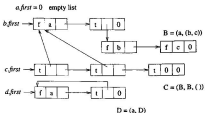


Figure 4.32: Representation of lists (1) to (4) (page 228); an *f* in the *flag* field represents the value false, whereas a *t* represents the value true

4.11.2 Recursive Algorithms for Lists

When a data object is defined recursively, it is often easy to describe recursive algorithms that work on these objects. Recursive algorithms for C++ objects typically consist of two components—the recursive function itself (the *workhorse*) and a second function that invokes the recursive function at the top level (the *driver*). The driver is declared as a public member function while the *workhorse* is declared as a private member function.

4.11.2.1 Copying a List

To see how recursion is useful, let us write a function (Program 4.35) that produces an exact copy of a nonrecursive list in which no sublists are shared.

```

// Driver
void GenList <T>::Copy(const GenList <T>& l) const
// Make a copy of l.
    first = Copy(l.first);
}

// Workhorse
GenListNode <T>* GenList <T>::Copy(GenListNode <T>* p)
// Copy the nonrecursive list with no shared sublists pointed at by p.
    GenListNode <T>* q = 0;
    if (p) {
        q = new GenListNode <T>;
        q->tag = p->tag;
        if (p->tag) q->down = Copy(p->down);
        else q->data = p->data;
        q->next = Copy(p->next);
    }
    return q;
}

```

Program 4.35: Copying a list

Program 4.35 reflects exactly the definition of a (generalized) list. We see immediately that *Copy* works correctly for an empty list. A simple proof using induction will verify the correctness of the entire function. Now let us consider the computing time of this function. The empty list takes a constant amount of time. For the list $A = ((a,b),(c,d),e)$, which has the representation of Figure 4.33, p takes on the values given in Figure 4.34. The sequence of values should be read down the columns; b, r, s, t, u, v, w , and x are the addresses of the eight nodes of the list. From this example one should be able to see that nodes with $\text{tag} = \text{false}$ will be visited twice, whereas nodes with $\text{tag} = \text{true}$ will be visited three times. Thus, if a list has a total of n nodes, no more than $3n$ executions of any statement will occur. Hence, the algorithm is $O(n)$, or linear, which is the best we can hope to achieve. Another factor of interest is the maximum depth of recursion or, equivalently, how many locations are needed for the recursion stack. Again, by carefully following the algorithm on the previous example, we

234 Linked Lists

see that the maximum depth is a combination of the lengths and depths of all sublists. However, a simple upper bound is m , the total number of nodes. Although this bound will be extremely large in many cases, it is achievable, for instance, if $A = (((((a))))))$.

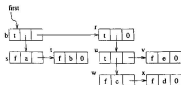


Figure 4.33: Linked representation for A

level of recursion	value of p	continuing level	p	continuing level	p
1	b	2	r	3	u
2	s	3	u	4	v
3	t	4	w	5	0
4	0	5	x	4	v
3	t	6	0	3	u
2	s	5	x	2	r
1	b	4	w	3	0
				2	r
				1	b

Figure 4.34 Values of parameters in execution of $GenList \langle T \rangle :: Copy(A)$

4.11.2.2 List Equality

Another useful function determines whether two lists are identical. To be identical, the lists must have the same structure and the same data in corresponding data members. Again, using the recursive definition of a list, we can write a short recursive function (Program 4.36) to accomplish this task.

```

// Driver
template <class T>
bool operator==(const GenList<T> & l) const
{ // *this and l are non-recursive lists.
  // The function returns true iff the two lists are identical.
    return Equal(l.first, l.first);
}

// Workhorse
bool Equal(GenListNode<T> *s, GenListNode<T> *t)
{
    if (!s) && (!t) return true;
    if (s && t && (s->tag == t->tag))
        if (s->tag)
            return Equal(s->down, t->down) && Equal(s->next, t->next);
        else return (s->data == t->data) && Equal(s->next, t->next);
    return false;
}

```

Program 4.36: Determining if two lists are identical

The computing time of Program 4.36 is clearly no more than linear when no sublists are shared, since it looks at each node of the two lists being compared no more than three times. For unequal lists the program terminates as soon as it discovers that the lists are not identical.

4.11.2.3 List Depth

Another handy operation on nonrecursive lists is the function that computes the depth of a list. The depth of the empty list is defined to be zero and, in general,

$$\text{depth}(s) = \begin{cases} 0 & \text{if } s \text{ is an atom} \\ 1 + \max \{ \text{depth}(x_1), \dots, \text{depth}(x_n) \} & \text{if } s \text{ is the list } (x_1, \dots, x_n), n \geq 1 \end{cases}$$

236 Linked Lists

Function `Depth` (Program 4.37) is a very close transformation of the definition, which is itself recursive.

```
// Driver
template <class T>
int GenList<T>::Depth()
// Compute the depth of a non-recursive list.
    return Depth(first);
}

// Workhorse
template <class T>
int GenList<T>::Depth(GenListNode<T> *x)
{
    if (!x) return 0; // empty list
    GenListNode<T> *current = x;
    int m = 0;
    while (current) {
        if (current->tag) m = max(m, Depth(current->down));
        current = current->next;
    }
    return m + 1;
}
```

Program 4.37: Computing the depth of a list

4.11.3 Reference Counts, Shared and Recursive Lists

In this section we shall consider some of the problems that arise when lists are allowed to be shared by other lists and when recursive lists are permitted. Sharing of sublists can, in some situations, result in great savings in storage used, as identical sublists occupy the same space. To facilitate specifying shared sublists, we extend the definition of a list to allow for naming of sublists. A sublist appearing within a list definition may be named through the use of a list name preceding it. For example, in the list $A = (a, (b, c))$, the sublist (b, c) could be assigned the name Z by writing $A = (a, Z(b, c))$. In fact, to be consistent, we would then write $A(a, Z(b, c))$ which would define the list A as above.

Lists that are shared by other lists, such as list B of Figure 4.32, create problems when you wish to add or delete a node at the front. If the first node of

B is deleted, it is necessary to change the pointers from list C to point to the second node. If a new node is added, pointers from C have to be changed to point to the new first node. However, we normally do not know all the pointers from which a particular list is being referenced. (Even if you did have this information, addition and deletion of nodes could require a large amount of time.) This problem is easily solved through the use of header nodes. If you expect to perform any additions and deletions at the front of lists, then the use of a header node with each list or named sublist will eliminate the need to retain a list of all pointers to any specific list. If each list is to have a header node, then lists (1) to (4) are represented as in Figure 4.35. The values in the *data/down* fields of the header nodes is the reference count (defined below) of the corresponding list. Even in situations in which you do not wish to add or delete nodes from lists dynamically, as in the case of multivariate polynomials, header nodes prove useful in determining when the nodes of a particular structure may be returned to the storage pool.

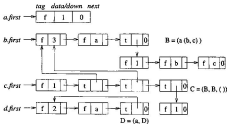


Figure 4.35: Structure with header nodes for lists (1) to (4)

Whenever lists are being shared by other lists, we need a mechanism to help determine whether or not the list nodes may be physically returned to the available-space list. This mechanism is generally provided through the use of a reference count maintained in the header node of each list. Since the *data* field of the header nodes is free, the reference count is maintained in this field.

238 Linked Lists

(Alternatively, a third variant may be introduced, with `tag` having three possible values: 0, 1, and 2.) The reference count of a list is the number of pointers (either program variables or pointers from other lists) to that list. The reference counts for the lists are

- (1) $a.first \rightarrow ref = 1$ accessible only via $a.first$
- (2) $b.first \rightarrow ref = 3$ pointed to by $b.first$ and two pointers from c
- (3) $c.first \rightarrow ref = 1$ accessible only via $c.first$
- (4) $d.first \rightarrow ref = 2$ accessible via $d.first$ and one pointer from itself.

Now a call to $c.GenList<T>()$ (list destructor) should result only in a decrementing by 1 of the reference counter of $i.first$. Only when the reference count becomes zero are the nodes of i deleted. The same is to be done with the sublists of c .

Suppose we change the definition of $GenListNode<T>$ to

```
template <class T>
class GenListNode <T>
{
friend class GenList <T>;
private:
    GenListNode <T> *next;
    int tag; // 0 for data, 1 for down, 2 for ref
    union {
        T data;
        GenListNode <T> *down;
        int ref;
    };
};
```

A recursive function to delete a list is given in Program 4.38. The workhorse proceeds by examining the top-level nodes of a list whose reference count has become zero. Any sublists encountered are deleted recursively, and finally, the top-level nodes are linked into the available-space list.

A call to $b.GenList<T>()$, where b is list (ii) of Figure 4.35, now has only the effect of decreasing the reference count of b to 2. Such a call followed by a call to $c.GenList<T>()$ results in

- (1) the reference count of c becomes zero
- (2) $b.first \rightarrow ref$ becomes 1 when the second top-level node of c is processed
- (3) $b.first \rightarrow ref$ becomes 0 when the third top-level node of c is processed;

```

// Driver
template <class T>
GenList <T>::GenList()
// Each header node has a reference count.
    if (first)
    {
        Delete (first);
        first = 0;
    }
}

// Workhorse
void GenList <T>::Delete(GenListNode <T>* x)
{
    x->ref--; // decrement reference count of header node.
    if (!x->ref)
    {
        GenListNode <T>* y = x; // y traverses top level of x.
        while (y->next) { y = y->next; if (y->tag == 1) Delete (y->down); }
        y->next = av; // attach top-level nodes to av list
        av = x;
    }
}

```

Program 4.38: Deleting a list recursively

now, the five nodes of list $B(a, (b, c))$ are returned to the available-space list

- (4) the top-level nodes of c are linked into the available-space list.

The use of header nodes with reference counts solves the problem of determining when nodes are to be physically freed in the case of shared sublists. However, for recursive lists, the reference count never becomes zero. $d.\text{GenList} <T>()$ results in $d.\text{first} \rightarrow \text{ref}$ becoming one. The reference count does not become zero, even though this list is no longer accessible either through program variables or through other structures. The same is true in the case of indirect recursion (Figure 4.36). After calls to $r.\text{GenList} <T>()$ and $s.\text{GenList} <T>()$, $r.\text{first} \rightarrow \text{ref} = 1$ and $s.\text{first} \rightarrow \text{ref} = 2$ but the structure consisting of r and s is no longer accessible. So, its nodes should have been returned to the available-space list.

Unfortunately, there is no simple way to supplement the list structure of

Figure 4.36c: Indirect recursion of lists $r = A$ and $s = A$

Figure 4.36 is to be able to determine when recursive lists may be physically deleted. It is no longer possible to return all free nodes to the available-space list when they become free. When recursive lists are being used, it is possible to run out of available space, even though not all nodes are in use.

EXERCISES

1. Describe how a multivariate polynomial may be represented using the following node structure that can be used to represent any generalized list:

tag = false/true	data/atom	next
------------------	-----------	------

2. Write a nonrecursive version of $GenList <T>::GenList <T>()$ (Program 4.38).
3. Write a nonrecursive version of `operator==` (Program 4.36).
4. Write a nonrecursive version of $GenList <T>::Depth$ (Program 4.37).
5. Write a function that inverts an arbitrary nonrecursive list l with no shared sublists. The sublists of l also are inverted. For example, if $l = (a, (b, c))$, then $invert(l) = ((c, b), a)$.
6. Derive a function that produces the list representation of an arbitrary list, given its linear form as a string of atoms, commas, blanks, and parentheses. For example, for the input $l = (a, (b, c))$, your function should produce the structure of Figure 4.37.

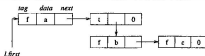


Figure 4.37: Structure for Exercise 6

7. One way to represent generalized lists is use nodes that have two fields plus a table that contains all atoms and list names, together with pointers to these lists. Let the two fields of each node be named *alink* and *blink*. Then *blink* either points to the next node on the same level, if there is one, or it is 0. The *alink* points either to a node at a lower level or, in the case of an atom or list name, to the appropriate entry in the symbol table. For example, the list $B(A, (D, E), ()B)$ would have the representation given in Figure 4.38. The notation d^* means a pointer to the first node of list D .

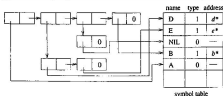


Figure 4.38: Representation for Exercise 7

(The list names D and E were already in the table at the time the list B was input. A was not in the table and is assumed to be an atom.)

242 Linked Lists

The symbol table retains a type bit for each entry. This bit is 1 if the entry is a list name and 0 for an atom. The NIL atom may be in the table, or *atom* can be set to 0 to represent the NIL atom. Write the C++ function `operator<>`, which inputs a list in parenthesis notation and sets up its linked representation as above. Note that no header nodes are in use. The following functions may be used by `operator<>`:

- (a) `int Get(s)` searches the symbol table for the name *s*. -1 is returned if *s* is not found in the table; otherwise, the position of *s* in the table is returned.
- (b) `Put(s,t,a)` enters the triple $\langle s, t, a \rangle$ into the table. If the name *s* is already in the table, then the type and address fields of the old entry are updated to *t* and *a*, respectively.
- (c) `NextToken()` gets the next token in the input list. (A token may be a list name, atom, '(', ')', or '.'. A '#' is returned if there are no more tokens.)

Write code for all functions used by you. You may assume that the input list is syntactically correct. If a sublist is labeled, as in the list $C(D, E(F, G))$, the structure should be set up as in the case $C(D, (F, G))$, and *E* should be entered into the symbol table as a list with the appropriate starting address.

- 8. What goes wrong when Program 4.35 is used to copy lists that have shared sublists?
- 9. Write and test a C++ template class for generalized lists with reference counts. You must include a copy constructor and destructor for your class as well as functions to determine the depth of a list and to determine whether two lists are identical. What is the time complexity of your functions?

CHAPTER 5

Trees

5.1 INTRODUCTION

5.1.1 Terminology

In this chapter we shall study a very important data object, the tree. Intuitively, a tree structure means that the data are organized in a hierarchical manner. One very common place where such a structure arises is in the investigation of genealogies. There are two types of genealogical charts that are used to present such data: the *pedigree* and the *lifest* chart. Figure 5.1 gives an example of each.

The pedigree chart of Figure 5.1(a) shows someone's ancestors, in this case those of Dusty, whose two parents are Honey Bear and Brandy. Brandy's parents are Coyote and Nugget, who are Dusty's grandparents on her father's side. The chart continues one more generation back to the great-grandparents. By the nature of things, we know that the pedigree chart is actually two-way branching, though this does not allow for inbreeding. When inbreeding occurs, we no longer have a tree structure unless we insist that each occurrence of

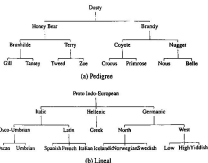


Figure 5.1: Two types of genealogical charts

breeding is separately listed. Inbreeding may occur frequently when describing family histories of flowers or animals.

The lineal chart of Figure 5.1(b), though it has nothing to do with people, is still a genealogy. It describes, in somewhat abbreviated form, the ancestry of the modern European languages. Thus, this is a chart of descendants rather than ancestors, and each item can produce several others. Latin, for instance, is the forebear of Spanish, French, and Italian. Proto Indo-European is a prehistoric language presumed to have existed in the fifth millennium B.C. This tree does not have the regular structure of the pedigree chart, but it is a tree structure nevertheless.

With these two examples as motivation, let us define formally what we mean by a tree.

Definition: A *tree* is a finite set of one or more nodes such that

- (1) There is a specially designated node called the *root*.
- (2) The remaining nodes are partitioned into a ≥ 0 disjoint sets T_1, \dots, T_n , where each of these sets is a tree. T_1, \dots, T_n are called the *subtrees* of the root. \square

Notice that this is a recursive definition. If we return to Figure 5.1, we see that the roots of the trees are Dusty and Proto Indo-European. Tree (a) has two subtrees, whose roots are Honey Bear and Brandy; tree (b) has three subtrees, with roots Italic, Hellenic, and Germanic. The condition that T_1, \dots, T_n be disjoint sets prohibits subtrees from ever connecting together (i.e., there is no cross-breeding). It follows that every item in a tree is the root of some subtree of the whole. For instance, Osco-Umbrian is the root of a subtree of Italic, which itself has two subtrees with the roots Oscan and Umbrian. Umbrian is the root of a tree with no subtrees.

There are many terms that are often used when referring to trees. A *node* stands for the item of information plus the branches to other nodes. Consider the tree in Figure 5.2. This tree has 13 nodes, each item of data being a single letter for convenience. The root is A, and we will normally draw trees with the root at the top.



Figure 5.2: A sample tree

The number of subtrees of a node is called its *degree*. The degree of A is 3, of C is 1, and of F is zero. Nodes that have degree zero are called *leaf* or *terminal* nodes. $\{K, L, F, G, M, I, J\}$ is the set of leaf nodes. Consequently, the other nodes are referred to as *nonterminal*. The roots of the subtrees of a node X are

the children of X . X is the parent of its children. Thus, the children of D are H , I , and J ; the parent of D is A . Children of the same parent are said to be siblings. H , I , and J are siblings. We can extend this terminology if we need to so that we can ask for the grandparent of M , which is D , and so on. The degree of a tree is the maximum of the degree of the nodes in the tree. The tree of Figure 5.2 has degree 3. The ancestors of a node are all the nodes along the path from the root to that node. The ancestors of M are A , D , and H .

The level of a node is defined by letting the root be at level one*. If a node is at level l , then its children are at level $l + 1$. Figure 5.2 shows the levels of all nodes in that tree. The height or depth of a tree is defined to be the maximum level of any node in the tree. Thus, the depth of the tree in Figure 5.2 is 4.

5.1.2 Representation of Trees

5.1.2.1 List Representation

There are several ways to draw a tree besides the one presented in Figure 5.2. One useful way is as a list. The tree of Figure 5.2 could be written as the list

$(A(B(E(K,L),F),C(G),D(H(M),I,J)))$

The information in the root node comes first, followed by a list of the subtrees of that node. This way of drawing trees leads to a memory representation of trees that is the same as that used for generalized lists in Chapter 4. Figure 5.3 shows the resulting memory representation for the tree of Figure 5.2. If we use this representation, we can make use of many of the general functions that we originally wrote for handling lists.

For several applications it is desirable to have a representation that is specialized to trees. One possibility is to represent each tree node by a memory node that has fields for the data and pointers to the tree node's children. Since the degree of each tree node may be different, we may be tempted to use memory nodes with a varying number of pointer fields. However, as it is often easier to write algorithms for a data representation when the node size is fixed, in practice one uses only nodes of a fixed size to represent tree nodes. For a tree of degree 4, we could use the node structure of Figure 5.4. Each child field is used to point to a subtree. Lemma 5.1 shows that using this node structure is very wasteful of space.

* Note that some authors define the level of the root to be 0.

Lemma 5.1
Kampt 95.11.98 V

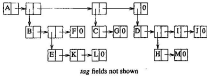


Figure 5.3: List representation of the tree of Figure 5.2

Figure 5.4: Possible node structure for a tree of degree k

Lemma 5.1: If T is a k -ary tree (i.e., a tree of degree k) with n nodes, each having a fixed size as in Figure 5.4, then $n(k-1) + 1$ of the nk child fields are 0, $n \geq 1$.

Proof: Since each non-zero child field points to a node and there is exactly one pointer to each node other than the root, the number of non-zero child fields in an n -node tree is exactly $n - 1$. The total number of child fields in a k -ary tree with n nodes is nk . Hence, the number of zero fields is $nk - (n - 1) = n(k - 1) + 1$. \square

We shall develop two specialized fixed-node-size representations for trees. Both of these require exactly two link, or pointer, fields per node.

5.1.1.2 Left Child-Right Sibling Representation

Figure 5.5 shows the node structure used in the left child–right sibling representation.



Figure 5.5: Left child–right sibling node structure

To convert the tree of Figure 5.2 into this representation, we first note that every node has at most one leftmost child and at most one closest right sibling. For example, in Figure 5.2, the leftmost child of *A* is *B*, and the leftmost child of *D* is *H*. The closest right sibling of *B* is *C*, and the closest right sibling of *H* is *I*. Strictly speaking, since the order of children in a tree is not important, any of the children of a node could be the leftmost child, and any of its siblings could be the closest right sibling. For the sake of definiteness, we choose the nodes based on how the tree is drawn. The *left child* field of each node points to its leftmost child (if any), and the *right sibling* field points to its closest right sibling (if any). Figure 5.6 shows the tree of Figure 5.2 redrawn using the left child–right sibling representation.



Figure 5.6: Left child–right sibling representation of tree of Figure 5.2

5.1.2.3 Representation as a Degree-Two Tree

To obtain the degree-two tree representation of a tree, we simply rotate the right-sibling pointers in a left child-right sibling tree clockwise by 45 degrees. This gives us the degree-two tree displayed in Figure 5.7. In the degree-two representation, we refer to the two children of a node as the left and right children. Notice that the right child of the root node of the tree is empty. This is always the case since the root of the tree we are transforming can never have a sibling. Figure 5.8 shows two additional examples of trees represented as left child-right sibling trees and as left child-right child (or degree-two) trees. Left child-right child trees are also known as *binary trees*.

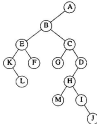


Figure 5.7: Left child-right child tree representation of tree of Figure 5.2

EXERCISES

1. Write a function to input a tree given as a generalized list (e.g., $(A(B(E(K,L),F),C(G),D(H(M),I,J)))$) and create its internal representation using nodes with three fields: *tag*, *data*, and *link*.

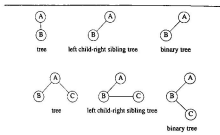


Figure 5.8: Tree representations

2. Write a function that reverses the process in Exercise 1 and takes a pointer to a tree and outputs it as a generalized list.
3. [Programming Project] Write the C++ class definition for trees using the list representation described in this section. Write the following C++ functions.
 - (a) `operator>>()`: accept a tree represented as a parenthesized list as input and create the generalized list representation of the tree (see Figure 5.3)
 - (b) `copy constructor`: initialize a tree with another tree represented as a generalized list
 - (c) `operator==()`: test for equality between two trees represented as generalized lists
 - (d) `destructor`: delete a tree represented as a generalized list
 - (e) `operator<<()`: output a tree in its parenthesized list notation

Test the correctness of your functions using suitable test data.

5.2 BINARY TREES

5.2.1 The Abstract Data Type

We have seen that we can represent any tree as a binary tree. In fact, binary trees are an important type of tree structure that occurs very often. Binary trees are characterized by the fact that any node can have at most two branches (i.e., there is no node with degree greater than two). For binary trees we distinguish between the subtree on the left and that on the right, whereas for trees the order of the subtrees is irrelevant. Also, a binary tree may have zero nodes. Thus, a binary tree is really a different object from a tree.

Definition: A *binary tree* is a finite set of nodes that either is empty or consists of a root and two disjoint binary trees called the *left subtree* and the *right subtree*. □

ADT 5.1 contains the specification for the binary tree data structure. This specification defines only a minimal set of operations on binary trees, which we use as a foundation on which to build additional operations.

Let us carefully review the distinctions between a binary tree and a tree. First, there is no tree having zero nodes, but there is an empty binary tree. Second, in a binary tree we distinguish between the order of the children; in a tree we do not. Thus, the two binary trees of Figure 5.9 are different, since the first binary tree has an empty right subtree, while the second has an empty left subtree. Viewed as trees, however, they are the same, despite the fact that they are drawn slightly differently.

Figure 5.10 shows two special kinds of binary trees. The first is a *skewed tree*, skewed to the left, and there is a corresponding tree that skews to the right. The tree of Figure 5.10(b) is called a *complete binary tree*. This kind of binary tree will be defined formally later. Notice that all leaf nodes are on adjacent levels. The terms that we introduced for trees such as *degree*, *level*, *height*, *leaf*, *parent*, and *child* all apply to binary trees in the natural way.

5.2.2 Properties of Binary Trees

Before examining data representations for binary trees, let us make some observations about such trees. In particular, we want to determine the maximum number of nodes in a binary tree of depth k and the relationship between the number of leaf nodes and the number of degree-two nodes in a binary tree.

```

template <class T>
class BinaryTree
{
// objects: A finite set of nodes either empty or consisting of a
// root node, left BinaryTree and right BinaryTree.
public:
    BinaryTree();
    // creates an empty binary tree

    bool IsEmpty();
    // return true iff the binary tree is empty

    BinaryTree(BinaryTree <T> & l, T & item, BinaryTree <T> & r);
    // creates a binary tree whose left subtree is l, whose right subtree
    // is r, and whose root node contains item

    BinaryTree <T> LeftSubtree();
    // return the left subtree of *this

    BinaryTree <T> RightSubtree();
    // return the right subtree of *this

    T RootData();
    // return the data in the root node of *this
};

```

ADT 5.1: Abstract data type *BinaryTree*

Figure 5.9: Two different binary trees

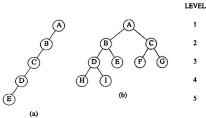


Figure 5.10: Skewed and complete binary trees

Lemma 5.2 (*Maximum number of nodes*):

- (1) The maximum number of nodes on level i of a binary tree is 2^{i-1} , $i \geq 1$.
- (2) The maximum number of nodes in a binary tree of depth k is $2^k - 1$, $k \geq 1$.

Proof:

- (1) The proof is by induction on i .

Induction Base: The root is the only node on level $i = 1$. Hence, the maximum number of nodes on level $i = 1$ is $2^{1-1} = 2^0 = 1$.

Induction Hypothesis: Let i be an arbitrary positive integer greater than 1. Assume that the maximum number of nodes on level $i - 1$ is 2^{i-2} .

Induction Step: The maximum number of nodes on level $i - 1$ is 2^{i-2} by the induction hypothesis. Since each node in a binary tree has a maximum degree of 2, the maximum number of nodes on level i is two times the maximum number of nodes on level $i - 1$, or 2^{i-1} .

(2) The maximum number of nodes in a binary tree of depth k is

$$\sum_{i=0}^k (\text{maximum number of nodes on level } i) = \sum_{i=0}^k 2^{i-1} = 2^k - 1 \quad \square$$

Lemma 5.3 [Relation between number of leaf nodes and degree-2 nodes]: For any nonempty binary tree, T , if n_0 is the number of leaf nodes and n_2 the number of nodes of degree 2, then $n_0 = n_2 + 1$.

Proof: Let n_1 be the number of nodes of degree one and n the total number of nodes. Since all nodes in T are at most of degree two, we have

$$n = n_0 + n_1 + n_2 \quad (5.1)$$

If we count the number of branches in a binary tree, we see that every node except the root has a branch leading into it. If B is the number of branches, then $n = B + 1$. All branches stem from a node of degree one or two. Thus, $B = n_1 + 2n_2$. Hence, we obtain

$$n = B + 1 = n_1 + 2n_2 + 1 \quad (5.2)$$

Subtracting Eq. (5.2) from Eq. (5.1) and rearranging terms, we get

$$n_0 = n_2 + 1 \quad \square$$

In Figure 5.10(a), $n_0 = 1$ and $n_2 = 0$; in Figure 5.10(b), $n_0 = 5$ and $n_2 = 4$.

We are now ready to define full and complete binary trees.

Definition: A *full binary tree* of depth k is a binary tree of depth k having $2^k - 1$ nodes, $k \geq 0$. \square

By Lemma 5.2, $2^k - 1$ is the maximum number of nodes in a binary tree of depth k . Figure 5.11 shows a full binary tree of depth 4. Suppose we number the nodes in a full binary tree starting with the root on level 1, continuing with the nodes on level 2, and so on. Nodes on any level are numbered from left to right. This numbering scheme gives us the definition of a complete binary tree.

Definition: A binary tree with n nodes and depth k is *complete* iff its nodes correspond to the nodes numbered from 1 to n in the full binary tree of depth k . \square

From Lemma 5.2, it follows that the height of a complete binary tree with n nodes is $\lceil \log_2(n + 1) \rceil$. (Note that $\lceil x \rceil$ is the smallest integer $\geq x$.)



Figure 5.11: Full binary tree of depth 4 with sequential node numbers

5.2.3 Binary Tree Representations

5.2.3.1 Array Representation

The numbering scheme used in Figure 5.11 suggests our first representation of a binary tree in memory. Since the nodes are numbered from 1 to n , we can use a one-dimensional array to store the nodes. (If the C++ array is used to represent the tree, the zero'th position is left empty.) Using Lemma 5.4 we can easily determine the locations of the parent, left child, and right child of any node, i , in the binary tree.

Lemma 5.4: If a complete binary tree with n nodes is represented sequentially, then for any node with index i , $1 \leq i \leq n$, we have

- (1) $\text{parent}(i)$ is at $\lfloor i/2 \rfloor$ if $i \neq 1$. If $i = 1$, i is at the root and has no parent.
- (2) $\text{leftChild}(i)$ is at $2i$ if $2i \leq n$. If $2i > n$, then i has no left child.
- (3) $\text{rightChild}(i)$ is at $2i + 1$ if $2i + 1 \leq n$. If $2i + 1 > n$, then i has no right child.

Proof: We prove (2). (3) is an immediate consequence of (2) and the numbering of nodes on the same level from left to right. (1) follows from (2) and (3). We prove (2) by induction on i . For $i = 1$, clearly the left child is at 2 unless $2 > n$, in which case i has no left child. Now assume that for all j , $1 \leq j \leq i$, $\text{leftChild}(j)$ is at $2j$. Then the two nodes immediately preceding $\text{leftChild}(i+1)$ are the right and left children of i . The left child is at $2i$. Hence, the left child of $i + 1$ is at $2i$

$+ 2 = 2(i + 1)$ unless $2(i + 1) > n$, in which case $i + 1$ has no left child. \square

This representation can clearly be used for all binary trees, though in most cases there will be a lot of unutilized space. Figure 5.12 shows the array representation for both trees of Figure 5.10. For complete binary trees such as the one in Figure 5.10(b), the representation is ideal, as no space is wasted. For the skewed tree of Figure 5.10(a), however, less than half the array is utilized. In the worst case a skewed tree of depth k will require $2^k - 1$ spaces. Of these, only k will be used.

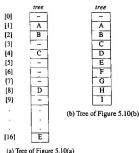


Figure 5.12: Array representation of the binary trees of Figure 5.10

5.2.3.2 Linked Representation

Although the array representation is good for complete binary trees, it is wasteful for many other binary trees. In addition, the representation suffers from the general inadequacies of sequential representations. Insertion and deletion of nodes

from the middle of a tree require the movement of potentially many nodes to reflect the change in level number of these nodes. These problems can be overcome easily through the use of a linked representation. Each node has three fields, *leftChild*, *data*, and *rightChild*. As with linked lists, we will use two classes to define a tree using a linked representation.

template <class T> class Tree; // forward declaration

```
template <class T>
class TreeNode {
friend class Tree <T>;
private:
    T data;
    TreeNode<T> *leftChild;
    TreeNode<T> *rightChild;
};
template <class T>
class Tree {
public:
    // Tree operations
private:
    TreeNode<T> *root;
};
```

We shall draw a tree node using either of the representations of Figure 5.13.



Figure 5.13: Node representations

Although with this node structure it is difficult to determine the parent of a node, we shall see that for most applications, this node structure is adequate. If it is necessary to be able to determine the parent of random nodes, then a fourth

258 Trees

field, *parent*, may be included in the class *TreeNode*. The representation of the binary trees of Figure 5.10 using this node structure is given in Figure 5.14. The root of the tree is stored in the data member *root* of *Tree*. This data member serves as the access pointer to the tree.

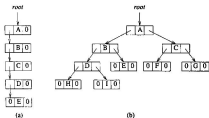


Figure 5.14: Linked representation for the binary trees of Figure 5.10

EXERCISES

1. For the binary tree of Figure 5.13, list the leaf nodes, the nonleaf nodes, and the level of each node.
2. What is the maximum number of nodes in a k -ary tree of height h ? Prove your answer.
3. Draw the internal memory representation of the binary tree of Figure 5.15 using (a) sequential and (b) linked representations.
4. Extend the array representation of a complete binary tree to the case of complete trees whose degree is d , $d > 1$. Develop formulas for the parent and children of the node stored in position i of the array.



Figure 5.15: Binary tree for Exercise 1

5.3 BINARY TREE TRAVERSAL AND TREE ITERATORS

5.3.1 Introduction

There are many operations that we often want to perform on trees. One notion that arises frequently is the idea of traversing a tree or visiting each node in the tree exactly once. When a node is visited, some operation (such as outputting its data field) is performed on it. A full traversal produces a linear order for the nodes in a tree. This linear order, given by the order in which the nodes are visited, may be familiar and useful. When traversing a binary tree, we want to treat each node and its subtrees in the same fashion. If we let *L*, *V*, and *R* stand for moving left, visiting the node, and moving right when at a node, then there are six possible combinations of traversal: *LVR*, *LRV*, *VLR*, *VRL*, *RVL*, and *RLV*. If we adopt the convention that we traverse left before right, then only three traversals remain: *LVR*, *LRV*, and *VLR*. To these we assign the names *inorder*, *postorder*, and *preorder*, respectively, because of the position of the *V* with respect to the *L* and the *R*. For example, in *postorder*, we visit a node after we have traversed its left and right subtrees, whereas in *preorder* the visiting is done before the traversal of these subtrees.

There is a natural correspondence between these traversals and producing the infix, postfix, and prefix forms of an expression. Consider the binary tree of Figure 5.16. This tree contains an arithmetic expression with the binary operators add (+), multiply (*), and divide (/) and the variables *A*, *B*, *C*, *D*, and *E*. For each node that contains an operator, its left subtree gives the left operand and its right subtree the right operand. We use this tree to illustrate each of the traversals.



Figure 5.16: Binary tree with arithmetic expression

5.3.2 Inorder Traversal

Informally, *inorder traversal* calls for moving down the tree toward the left until you can go no farther. Then you “visit” the node, move one node to the right and continue. If you cannot move to the right, go back one more node. A precise way of describing this traversal is by using recursion as in Program 5.1.

Recursion is an elegant device for describing this traversal. Figure 5.17 is a trace of how function `inorder(TreeNode<T> *)` (Program 5.1) works on the tree of Figure 5.16. Read down the left column first and then the right one. Figure 5.17 assumes that the `Visit` function has a single line of code:

```
cout << currentNode->data;
```

The elements get output in the following order:

$$A / B * C * D + E$$

which is the infix form of the expression.

5.3.3 Preorder Traversal

The C++ code for the second form of traversal, *preorder*, is given in Program 5.2. In words, we would say “visit a node, traverse left, and continue. When

```

1 template <class T>
2 void Tree <T>::Inorder()
3 // Driver calls workhorse for traversal of entire tree. The driver is
4 // declared as a public member function of Tree.
5     Inorder(root);
6 }

7 template <class T>
8 void Tree <T>::Inorder(TreeNode <T> *currentNode)
9 // Workhorse traverses the subtree rooted at currentNode.
10 // The workhorse is declared as a private member function of Tree.
11     if (currentNode) {
12         Inorder(currentNode->leftChild);
13         Visit (currentNode);
14         Inorder (currentNode->rightChild);
15     }
16 }

```

Program 5.1: Inorder traversal of a binary tree

you cannot continue, move right and begin again or move back until you can move right and resume.” The nodes of Figure 5.16 would be output in *preorder* as

$$+ * + / A B C D E$$

which we recognize as the *prefix* form of the expression.

5.3.4 Postorder Traversal

The code for *postorder* traversal is given in Program 5.3. On the tree of Figure 5.16, this function produces the following output:

$$A B / C * D * E +$$

which is the *postfix* form of our expression.

Call of inorder	Value in currentNode	Action	Call of inorder	Value in currentNode	Action
Driver	+		10	C	
1	+		11	0	
2	+		10	C	cout << 'C'
3	/		12	0	
4	A		1	+	cout << 'A'
5	0		13	D	
4	A	cout << 'A'	14	0	
6	0		13	D	cout << 'D'
3	/	cout << '/'	15	0	
7	B		Driver	+	cout << 'A'
8	0		16	E	
7	B	cout << 'B'	17	0	
9	0		16	E	cout << 'E'
2	+	cout << '+'	18	0	

Figure 5.17: Trace of Program 5.1

5.3.5 Iterative Inorder Traversal

Since the tree is a container class, we would like to implement an iterator for the class *Tree*. Suppose that we have decided that our iterator will sequence through the elements in the tree in inorder. To implement such an inorder iterator, we first need to implement the inorder traversal method without using recursion. A nonrecursive code for inorder traversal is given in Program 5.4. This program USES-A stack as defined in ADT 3.1.

Definition: We say that a data object of Type *X* USES-A data object of Type *Y* if a Type *X* object uses a Type *Y* object to perform a task. This relationship is typically expressed by employing the Type *Y* object in a member function of Type *X*. The Type *Y* object may be passed as an argument to the member function or used as a local variable in the function. The USES-A relationship is similar to the IS-IMPLEMENTED-IN-TERMS-OF relationship. The difference is that the degree to which the Type *Y* object is used is less in the USES-A relationship. □

For our purposes, the stack template of ADT 3.1 is instantiated to *DecNode<T>*.

Analysis of NonrecInorder: Let *n* be the number of nodes in the tree. If we consider the action of Program 5.4, we note that every node of the tree is placed

```

1 template <class T>
2 void Tree <T>::Preorder()
3 { // Driver.
4     Preorder(root);
5 }

6 template <class T>
7 void Tree <T>::Preorder(TreeNode <T> *currentNode)
8 { // Workhorse.
9     if (currentNode) {
10         Visit(currentNode);
11         Preorder(currentNode->leftChild);
12         Preorder(currentNode->rightChild);
13     }
14 }

```

Program 5.2: Preorder traversal of a binary tree

```

1 template <class T>
2 void Tree <T>::Postorder()
3 { // Driver.
4     Postorder(root);
5 }

6 template <class T>
7 void Tree <T>::Postorder(TreeNode <T> *currentNode)
8 { // Workhorse.
9     if (currentNode) {
10         Postorder(currentNode->leftChild);
11         Postorder(currentNode->rightChild);
12         Visit(currentNode);
13     }
14 }

```

Program 5.3: Postorder traversal of a binary tree

```

1 template <class T>
2 void Tree<T>::Nonrecursive()
3 { // Nonrecursive inorder traversal using a stack.
4   Stack<TreeNode<T>*> s; // declare and initialize stack
5   TreeNode<T> *currentNode = root;
6   while(1) {
7     while (currentNode) { // move down leftChild fields
8       s.Push(currentNode); // add to stack
9       currentNode = currentNode->leftChild;
10    }
11    if (s.isEmpty()) return;
12    currentNode = s.Top();
13    s.Pop(); // delete from stack
14    Visit(currentNode);
15    currentNode = currentNode->rightChild;
16  }
17 }

```

Program 5.4: Nonrecursive inorder traversal

on the stack once. Thus, the statements on lines 8, 9 and 11 to 15 are executed n times. Moreover, `currentNode` will equal 0 once for every 0 link in the tree, which is exactly

$$2n_0 + n_1 = n_0 + n_1 + n_2 + 1 = n + 1$$

Every step will be executed no more than some constant times n , so the time complexity is $O(n)$. The run time can be reduced by a constant factor by eliminating some of the unnecessary stacking (see Exercises). The space required for the stack is equal to the depth of the tree. This is at most n . \square

We now use the function `Nonrecursive` to obtain an inorder iterator for a tree. Rather than develop an iterator as intricate as a C++ forward iterator, we consider a simple iterator that has only a `Next` function, which returns a pointer to the next element in inorder. The key observation required to develop our simplified iterator is that each iteration of the while loop of lines 6–16 in Program 5.4 yields the next element in the inorder traversal of the tree. We begin by defining the class `InorderIterator`, which is a nested class (and a friend) of `Tree`. The data members required for this class are the two objects `s` and `currentNode` as used in Program 5.4. Program 5.5 contains the class definition of

InorderIterator. The constructor for this class initializes *currentNode* to the tree root. The code implementing *Next()* is obtained by extracting lines 7-15 of Program 5.4 corresponding to a single iteration of the while loop. Instead of visiting the next element, we return this element. Program 5.6 gives the resulting code.

```
class InorderIterator {
public:
    InorderIterator()(currentNode = root);
    T * Next();
private:
    Stack<TreeNode<T>*> s;
    TreeNode<T> *currentNode;
};
```

Program 5.5: Definition of a simple inorder iterator class

```
T *InorderIterator::Next()
{
    while (currentNode) {
        s.Push(currentNode);
        currentNode = currentNode->leftChild;
    }
    if (s.IsEmpty()) return 0;
    currentNode = s.Top();
    s.Pop();
    T& temp = currentNode->data;
    currentNode = currentNode->rightChild;
    return &temp;
}
```

Program 5.6: Code for obtaining the next inorder element

5.3.6 Level-Order Traversal

Whether written iteratively or recursively, the inorder, preorder, and postorder traversals all require a stack. We now turn to a traversal that requires a queue.

This traversal, called *level-order traversal*, visits the nodes using the ordering suggested by the node numbering scheme of Figure 5.11. Thus, we visit the root first, then the root's left child, followed by the root's right child. We continue in this manner, visiting the nodes at each new level from the leftmost node to the rightmost node.

The code for this traversal, Program 5.7, uses the template class *Queue* of ADT 3.2.

```

template <class T>
void Trer <T>::LevelOrder ()
// Traverse the binary tree in level order.
{
    Queue <TreeNode <T>*> q;
    TreeNode <T> *currentNode = root;
    while (currentNode) {
        Visit (currentNode);
        if (currentNode->leftChild) q.Push (currentNode->leftChild);
        if (currentNode->rightChild) q.Push (currentNode->rightChild);
        if (q.IsEmpty ()) return;
        currentNode = q.Front ();
        q.Pop ();
    }
}

```

Program 5.7: Level-order traversal of a binary tree

We begin by visiting the root and adding its children to the queue. The next node to visit is obtained from the front of the queue. Since a node's children are at the next lower level, and we add the left child before the right child, the nodes are output using the ordering scheme found in Figure 5.11. The level-order traversal of the tree in Figure 5.16 is

+ * E * D / C A B

5.3.7 Traversal without a Stack

Before we leave the topic of tree traversal, we shall consider one final question. Is binary tree traversal possible without the use of extra space for a stack? (Note that a recursive tree traversal algorithm also implicitly uses a stack.) One simple solution is to add a *parent* field to each node. Then we can trace our way back up to any root and down again. Another solution, which requires two bits per

node, represents binary trees as threaded binary trees. We study this in Section 5.5. If the allocation of this extra space is too costly, then we can use the *leftChild* and *rightChild* fields to maintain the paths back to the root. The stack of addresses is stored in the leaf nodes. The exercises examine this algorithm more closely.

EXERCISES

1. Write out the inorder, preorder, postorder, and level-order traversals for the binary trees of Figure 5.10.
2. Do Exercise 1 for the binary tree of Figure 5.11.
3. Do Exercise 1 for the binary tree of Figure 5.15.
4. Implement a forward iterator for *Tree*. Your iterator should traverse the tree in inorder.
5. Implement a forward iterator for *Tree*. Your iterator should traverse the tree in level order.
6. Write a nonrecursive version of function *Preorder* (Program 5.2).
7. Use the results of the previous exercise to implement a forward iterator, *PreorderIterator*, that traverse the tree in preorder.
8. Write a nonrecursive version of function *Postorder* (Program 5.3).
9. Use the results of the previous exercise to implement a forward iterator, *PostorderIterator*, that traverse the tree in postorder.
10. [Programming Project]: Develop a complete C++ template class for binary trees. You must include a constructor, copy constructor, destructor, the four traversal methods of this section together with forward iterators for each. Include also the remaining functions specified in ADT 5.1.
11. Rework *NonrecursiveOrder* (Program 5.4) so that it is as fast as possible. (Hint: Minimize the stacking and the testing within the loop.)
12. Program 5.8 performs an inorder traversal without using a stack. Verify that the code is correct by running it on a variety of binary trees that cause every statement to execute at least once.

```
template <class T>
void Tree<T>::NoStackInorder()
// Inorder traversal of binary tree using a fixed amount of additional storage.
if (!root) return; // empty binary tree
TreeNode<T> *top = 0, *lastRight = 0, *p, *q, *r, *r1;
p = q = root;
while (1) {
```

```

while (1) {
    if (!p → leftChild) && (!p → rightChild) { // leaf node
        Visit(p); break;
    }
    if (!p → leftChild) { // visit p and move to p → rightChild
        Visit(p);
        r = p → rightChild; p → rightChild = q;
        q = p; p = r;
    }
    else { // move to p → leftChild
        r = p → leftChild; p → leftChild = q;
        q = p; p = r;
    }
} // end of inner while
// p is a leaf node, move upward to a node whose
// right subtree has not yet been examined
TreeNode <T> *av = p;
while (1) {
    if (p == root) return;
    if (!q → leftChild) { // q is linked via rightChild
        r = q → rightChild; q → rightChild = p;
        p = q; q = r;
    }
    else if (!q → rightChild) { // q is linked via leftChild
        r = q → leftChild; q → leftChild = p;
        p = q; q = r; Visit(p);
    }
    else // check if p is a rightChild of q
        if (q == lastRight) {
            r = top; lastRight = r → leftChild;
            top = r → rightChild; // unstack
            r → leftChild = r → rightChild = 0;
            r = q → rightChild; q → rightChild = p;
            p = q; q = r;
        }
    else { // p is leftChild of q
        Visit(q);
        av → leftChild = lastRight; av → rightChild = top;
        top = av; lastRight = q;
        r = q → leftChild; q → leftChild = p; // restore link to p
        r1 = q → rightChild; q → rightChild = r;
        p = r1;
    }
}

```

```

        break;
    }
} // end of inner while loop
} // end of outer while loop
}

```

Program 5.8: $O(1)$ space inorder traversal

13. Write a nonrecursive version of *Postorder* (Program 5.3) using only a fixed amount of additional space. (Use the ideas of the previous exercise.)
14. Do the preceding exercise for the case of *Preorder* (Program 5.2).

5.4 ADDITIONAL BINARY TREE OPERATIONS

5.4.1 Copying Binary Trees

Using the definition of a binary tree and the recursive version of the traversals, we can easily write other routines for working with binary trees. For instance, if we want to implement a copy constructor to initialize a binary tree with an exact copy of another binary tree, we can modify the postorder traversal algorithm only slightly to get Program 5.9, which assumes that *TreeNode* has a constructor that sets all three data members of a tree node.

5.4.2 Testing Equality

Another problem that is especially easy to solve using recursion is determining the equivalence of two binary trees. Binary trees are equivalent if they have the same topology and the information in corresponding nodes is identical. By the same topology we mean that every branch in one tree corresponds to a branch in the second in the same order and vice versa. The function operator `==()` calls the workhorse function *Equal* (Program 5.10), which traverses the binary trees in preorder, though any order could be used.

5.4.3 The Satisfiability Problem

Consider the set of formulas we can construct by taking variables x_1, x_2, x_3, \dots , and the operators \wedge (and), \vee (or), and \neg (not). These variables can hold only one of two possible values, *true* or *false*. The set of expressions that can be

```

template <class T>
Tree<T>::Tree(const Tree<T>& t) // driver
{ // Copy constructor
    root = Copy(t.root);
}

template <class T>
TreeNode<T>* Tree<T>::Copy(TreeNode<T>* origNode) // Workhorse
{ // Return a pointer to an exact copy of the binary tree rooted at origNode.
    if (!origNode) return 0;
    return new TreeNode<T>(origNode->data,
                           Copy(origNode->leftChild),
                           Copy(origNode->rightChild));
}

```

Program 5.9: Copying a binary tree

```

template <class T>
bool Tree<T>::operator==(const Tree& t) const
{
    return Equal(root,t.root);
}

template <class T>
bool Tree<T>::Equal(TreeNode<T>* a, TreeNode<T>* b)
{ // Workhorse.
    if (!a) && (!b) return true; // both a and b are 0
    return (a && b // both a and b are non-zero
           && (a->data == b->data) // data is the same
           && Equal(a->leftChild,b->leftChild) // left subtrees equal
           && Equal(a->rightChild,b->rightChild)); // right subtrees equal
}

```

Program 5.10: Binary tree equivalence

formed using these variables and operators is defined by the following rules:

- (1) a variable is an expression
- (2) if x and y are expressions then $x \wedge y$, $x \vee y$, and $\neg x$ are expressions
- (3) parentheses can be used to alter the normal order of evaluation, which is not before and before or.

This set defines the formulas of the *propositional calculus* (other operations such as implication can be expressed using \wedge , \vee , and \neg). The expression

$$x_1 \vee (x_2 \wedge \neg x_3)$$

is a formula (read “ x_1 or x_2 and not x_3 ”). If x_1 and x_3 are *false* and x_2 is *true*, then the value of this expression is

$$\begin{aligned} & \text{false} \vee (\text{true} \wedge \neg \text{false}) \\ &= \text{false} \vee \text{true} \\ &= \text{true} \end{aligned}$$

The *satisfiability problem* for formulas of propositional calculus asks if there is an assignment of values to the variables that causes the value of the expression to be true.

Again, let us assume that our formula is already in a binary tree, say

$$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2) \vee \neg x_3$$

in the tree of Figure 5.18. The inorder traversal of this tree is

$$x_1 \wedge \neg x_2 \vee \neg x_1 \wedge x_2 \vee \neg x_3$$

which is the infix form of the expression. The most obvious algorithm to determine satisfiability is to let (x_1, x_2, x_3) take on all possible combinations of *true* and *false* values and to check the formula for each combination. For n variables there are 2^n possible combinations of *true* = t and *false* = f . For example, for $n = 3$, the eight combinations are: (t, t, t) , (t, t, f) , (t, f, t) , (t, f, f) , (f, t, t) , (f, t, f) , (f, f, t) , (f, f, f) . The algorithm will take $O(g2^n)$, or exponential time, where g is the time to substitute values for x_1, x_2, \dots, x_n and evaluate the expression.

To evaluate an expression, we traverse its tree in postorder. When visiting a node p , we compute the value of the expression represented by the subtree rooted at p . Recall that, in postorder, the left and right subtrees of a node are traversed before we visit that node. In other words, when we visit the node p , the subexpressions represented by its left and right subtrees have been computed. So, when we reach the \vee node on level 2, the values of $x_1 \wedge \neg x_2$ and $\neg x_1 \wedge x_2$ will already be available to us, and we can apply the rule for or. Notice that a node containing \neg has only a right branch, since \neg is a unary operator.



Figure 5.18: Propositional formula in a binary tree

For our satisfiability problem application, we shall instantiate the template class *Tree* with $T = \text{pair} < \text{Operator}, \text{bool} >$, where *pair* is the predefined C++ template structure with two data members *first* and *second*. The data types of these two data members are, respectively, *Operator* and *bool*. The data type *Operator* is defined by us as below.

```
enum Operator {Not, And, Or, True, False};
```

Strictly speaking, *True* and *False* denote constants rather than operators. The first version of our algorithm for the satisfiability problem is Program 5.11. In this, n is the number of variables in the formula and *formula* is the binary tree that represents the formula.

Program 5.12 provides the code for tasks to be performed when visiting a node of the expression tree. For simplicity, Program 5.12 assumes that for every leaf node, its *data.first* field has been set either to *True* or *False* depending on the current truth assignment of the variable for that leaf node.

EXERCISES

1. Write a C++ function to count the number of leaf nodes in a binary tree. What is its computing time?

```

for each of the  $2^n$  possible truth value combinations for the  $n$  variables
{
    replace the variables by their values in the current truth value combination;
    evaluate the formula by traversing the tree it points to in postorder;
    if (formula.Data().second()) { cout << "current combination"; return;}
}
cout << "no satisfiable combination";

```

Program 5.11: First version of satisfiability algorithm

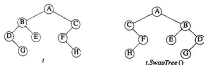
```

// visit the node pointed at by p
switch (p->data.first) {
    case Not: p->data.second = !p->rightChild->data.second; break;
    case And: p->data.second =
        p->leftChild->data.second && p->rightChild->data.second;
        break;
    case Or: p->data.second =
        p->leftChild->data.second || p->rightChild->data.second;
        break;
    case True: p->data.second = true; break;
    case False: p->data.second = false;
}

```

Program 5.12: Visiting a node in an expression tree

2. Write a C++ function, *SwapTree()*, that swaps the left and right children of every node of a binary tree. An example is given in Figure 5.19.
3. Devise an external representation for the formulas in propositional calculus. Write a function that reads such a formula and creates its binary tree representation. What is the complexity of your function?
4. [Destructor] Write a recursive function to delete all nodes in a binary tree. What is the complexity of your function?
5. Write a recursive function to assign one tree to another (operator=). What is the complexity of your function?

Figure 5.19: A *SwapTree* example

5.5 THREADED BINARY TREES

5.5.1 Threads

If we look carefully at the linked representation of any binary tree, we notice that there are more 0-links than actual pointers. As we saw before, there are $n + 1$ 0-links and $2n$ total links. A clever way to make use of these 0-links has been devised by A. J. Perlis and C. Thornton. Their idea is to replace the 0-links by pointers, called threads, to other nodes in the tree. These threads are constructed using the following rules:

- (1) A 0 *rightChild* field in node p is replaced by a pointer to the node that would be visited after p when traversing the tree in inorder. That is, it is replaced by the inorder successor of p .
- (2) A 0 *leftChild* link at node p is replaced by a pointer to the node that immediately precedes node p in inorder (i.e., it is replaced by the inorder predecessor of p).

Figure 5.20 shows the binary tree of Figure 5.10(b) with its new threads drawn in as broken lines. This tree has 9 nodes and 10 0-links, which have been replaced by threads. If we traverse the tree in inorder, the nodes will be visited in the order $H, D, I, R, E, A, F, C, G$. For example, node E has a predecessor thread that points to R and a successor thread that points to A .

In the memory representation we must be able to distinguish between threads and normal pointers. This is done by adding two Boolean fields, *leftThread* and *rightThread*, to *TreeNode*. Let t be a pointer to a tree node. If $t \rightarrow \text{leftThread} = \text{true}$, then $t \rightarrow \text{leftChild}$ contains a thread; otherwise it contains a pointer to the left child. Similarly if $t \rightarrow \text{rightThread} = \text{true}$, then

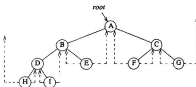


Figure 5.20: Threaded tree corresponding to Figure 5.10(b)

$t \rightarrow \text{rightChild}$ contains a thread; otherwise it contains a pointer to the right child. Let *ThreadedNode* be the resulting node structure.

In Figure 5.20 we see that two threads have been left dangling. One is the *leftChild* of *H* and the other the *rightChild* of *G*. In order that we leave no loose threads, we will assume a header node for all threaded binary trees. The original tree is the left subtree of the header node. An empty binary tree is represented by its header node as in Figure 5.21. The complete memory representation for the tree of Figure 5.20 is shown in Figure 5.22.

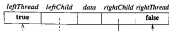


Figure 5.21: An empty threaded binary tree

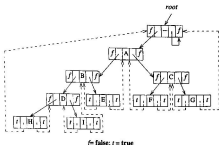


Figure 5.22: Memory representation of threaded tree

5.5.1 Inorder Traversal of a Threaded Binary Tree

By using the threads, we can perform an inorder traversal without making use of a stack. Observe that for any node x in a binary tree, if $x \rightarrow \text{rightThread} == \text{true}$, then the inorder successor of x is $x \rightarrow \text{rightChild}$ by definition of threads. Otherwise the inorder successor of x is obtained by following a path of left-child links from the right child of x until a node with $\text{leftThread} == \text{true}$ is reached. Function *Next()* (Program 5.13) finds and returns the inorder successor of node *currentNode* in a threaded binary tree. (We assume that *ThreadedInorderTree* is a nested class of *ThreadedTree*.)

The interesting thing to note about function `Next()` is that it is now possible to find the inorder successor of any arbitrary node in a threaded binary tree without using an additional stack. Since the tree is the left subtree of the header node and because of the choice of `rightThreaded = false` for the header node, the inorder sequence of nodes in the original binary tree may be obtained by repeated invocations of the function `Next()` beginning with `currentNode` pointing to the tree header. The computing time for the traversal is readily seen to be

```

T* ThreadedInorderIterator::Next()
{
    // Return the inorder successor of currentNode in a threaded binary tree
    ThreadedNode<T> *temp = currentNode->rightChild;
    if (!currentNode->rightThread)
        while (!temp->leftThread) temp = temp->leftChild;
    currentNode = temp;
    if (currentNode == root) return 0;
    else return &currentNode->data;
}

```

Program 5.13: Finding the inorder successor in a threaded binary tree

$O(n)$ for an n -node tree.

5.5.3 Inserting a Node into a Threaded Binary Tree

We now examine how to make insertions into a threaded tree. This will give us a function for growing threaded trees. We shall study only the case of inserting r as the right child of a node s . The case of insertion of a left child is given as an exercise. The cases for insertion are

- (1) If s has an empty right subtree, then the insertion is simple and diagrammed in Figure 5.23(a).
- (2) If the right subtree of s is not empty, then this right subtree is made the right subtree of r after insertion. When this is done, r becomes the inorder predecessor of a node that has a `leftThread == true` field, and consequently there is a thread which has to be updated to point to r . The node containing this thread was previously the inorder successor of s . Figure 5.23(b) illustrates the insertion for this case.

In both cases s is the inorder predecessor of r . The details are given in function *InsertRight* (Program 5.14). It is assumed that function *InorderSucc*(r) returns the inorder successor of r , using an algorithm similar to that used in *Next*().

EXERCISES

1. Write a C++ function to insert a new node l as the left child of node s in a threaded binary tree. The left subtree of s becomes the left subtree of l .

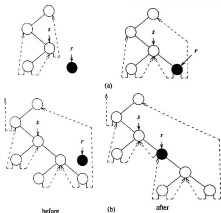


Figure 5.23: Insertion of r as a right child of s in a threaded binary tree

2. Write a C++ forward iterator for threaded binary trees. You should sequence through the nodes in inorder.
3. Write a function to traverse a threaded binary tree in postorder. What are the time and space requirements of your function?
4. Write a function to traverse a threaded binary tree in preorder. What are the time and space requirements of your method?
5. Consider threading a binary tree using preorder threads rather than inorder threads as in the text. Which of the traversals can be done without the use of a stack? For those that can be performed without a stack, write a C++

```

template <class T>
void ThreadedTree <T>::InsertRight(ThreadedNode <T> *r,
                                   ThreadedNode <T> *r1)
{ // Insert r as the right child of r1.
  r->rightChild = r1->rightChild;
  r->rightThread = r1->rightThread;
  r->leftChild = r1;
  r->leftThread = true; // leftChild is a thread
  r1->rightChild = r;
  r1->rightThread = false;
  if (! r->rightThread) {
    ThreadedNode <T> *temp = InorderSucc(r1);
                                // returns the inorder successor of r1
    temp->leftChild = r;
  }
}

```

Program 5.14: Inserting r as the right child of r_1

function and analyze its space complexity.

6. Consider threading a binary tree using postorder threads rather than inorder threads as in the text. Which of the traversals can be done without the use of a stack? For those that can be performed without a stack, write an algorithm and analyze its space complexity.

5.6 HEAPS

5.6.1 Priority Queues

Heaps are frequently used to implement *priority queues*. In this kind of queue, the element to be deleted is the one with highest (or lowest) priority. At any time, an element with arbitrary priority can be inserted into the queue. ADT 5.2 specifies a max priority queue as a C++ abstract class.

It is assumed that the type T is defined so that the C++ relational operators ($<$, $>$, etc.) compare element priorities. We represent $MaxPQ$ using pure virtual functions (and hence make it an abstract class) because it can be realized by a number of different data structures. Without knowing which data structure will be used, it is impossible to implement the $MaxPQ$ operations. However, we know that any data structure D that implements a max priority queue must

```

template <class T>
class MaxPQ {
public:
    virtual ~MaxPQ() {}
        // virtual destructor
    virtual bool IsEmpty() const = 0;
        // returns true iff the priority queue is empty
    virtual const T& Top() const = 0;
        // return reference to max element
    virtual void Push(const T&) = 0;
        // add an element to the priority queue
    virtual void Pop() = 0;
        // delete element with max priority
};

```

ADT 5.2: A max priority queue

implement the operations specified in ADT 5.2. This is ensured by implementing D as a publicly derived class of $MaxPQ$.

Example 5.1: Suppose that we are selling the services of a machine. Each user pays a fixed amount per use. However, the time needed by each user is different. We wish to maximize the returns from this machine under the assumption that the machine is not to be kept idle unless no user is available. This can be done by maintaining a priority queue of all persons waiting to use the machine. Whenever the machine becomes available, the user with the smallest time requirement is selected. Hence, a min priority queue is required. When a new user requests the machine, his/her request is put into the priority queue.

If each user needs the same amount of time on the machine but people are willing to pay different amounts for the service, then a priority queue based on the amount of payment can be maintained. Whenever the machine becomes available, the user paying the most is selected. This requires a max priority queue. \square

Example 5.2: Suppose that we are simulating a large factory. This factory has many machines and many jobs that require processing on some of the machines. An event is said to occur whenever a machine completes the processing of a job. When an event occurs, the job has to be moved to the queue for the next machine (if any) that it needs. If this queue is empty, the job can be assigned to the machine immediately. Also, a new job can be scheduled on the machine that has

become idle (provided that its queue is not empty).

To determine the occurrence of events, a priority queue is used. This queue contains the finish time of all jobs that are presently being worked on. The next event occurs at the least time in the priority queue. So, a min priority queue can be used in this application. \square

The simplest way to represent a priority queue is as an unordered linear list. Regardless of whether this list is represented sequentially or as a chain, the *IsEmpty* function takes $O(1)$ time; the *Top()* function takes $O(n)$ time, where n is the number of elements in the priority queue; a push can be done in $O(1)$ time as it doesn't matter where in the list the new element is inserted; and a *Pop* takes $O(n)$ time as one must first find the element with max priority and then delete it. As we shall see shortly, when a max heap is used, the complexity of *IsEmpty* and *Top* is $O(1)$ and that of *Push* and *Pop* is $O(\log n)$.

5.6.2 Definition of a Max Heap

In Section 5.2.2, we defined a complete binary tree. In this section we present a special form of a complete binary tree that is useful in many applications.

Definition: A *max (min) tree* is a tree in which the key value in each node is no smaller (larger) than the key values in its children (if any). A *max heap* is a complete binary tree that is also a max tree. A *min heap* is a complete binary tree that is also a min tree. \square

Some examples of max heaps and min heaps are shown in Figures 5.24 and 5.25, respectively.



Figure 5.24: Max heaps

From the definitions, it follows that the key in the root of a min tree is the smallest key in the tree, whereas that in the root of a max tree is the largest.



Figure 5.25: Min heaps

When viewed as an ADT, a max heap is very simple. The basic operations are the same as those for a max priority queue (ADT 5.2). Since a max heap is a complete binary tree, we represent it using an array heap. The private data members of the template class *MaxHeap*, which derives from *MaxPQ*<*T*>, are given below.

private:

<i>T</i> * <i>heap</i> ;	// element array
int <i>heapSize</i> ;	// number of elements in heap
int <i>capacity</i> ;	// size of the array heap

Program 5.15 defines the *MaxHeap* constructor. A max heap is empty iff *heapSize* == 0; the max element of a nonempty max heap is in *heap*[1]. Since the codes for *IsEmpty* and *Top* are fairly straightforward, these are omitted.

```

template <class T>
MaxHeap<T>::MaxHeap (int theCapacity = 10)
{
    if (theCapacity < 1) throw "Capacity must be >= 1.";
    capacity = theCapacity;
    heapSize = 0;
    heap = new T [capacity+1]; // heap[0] is not used
}
  
```

Program 5.15: Max heap constructor

5.6.3 Insertion into a Max Heap

A max heap with five elements is shown in Figure 5.26(a). When an element is added to this heap, the resulting six-element heap must have the structure shown in Figure 5.26(b), because a heap is a complete binary tree. To determine the correct place for the element that is being inserted, we use a *bubbling up* process that begins at the new node of the tree and moves toward the root. The element to be inserted bubbles up as far as is necessary to ensure a max heap following the insertion. If the element to be inserted has key value 1, it may be inserted as the left child of 2 (i.e., in the new node). If instead, the key value of the new element is 5, then this cannot be inserted as the left child of 2 (as otherwise, we will not have a max heap following the insertion). So, the 2 is moved down to its left child (Figure 5.26(c)), and we determine if placing the 5 at the old position of 2 results in a max heap. Since the parent element (20) is at least as large as the element being inserted (5), it is all right to insert the new element at the position shown in the figure. Next, suppose that the new element has value 21 rather than 5. In this case, the 2 moves down to its left child as in Figure 5.26(c). The 21 cannot be inserted into the old position occupied by the 2, as the parent of this position is smaller than 21. Hence, the 20 is moved down to its right child and the 21 inserted into the root of the heap (Figure 5.26(d)).

To implement the insertion strategy just described, we need to go from an element to its parent. Lemma 5.4 enables us to locate the parent of any element easily. Program 5.16 performs an insertion into a max heap.

Analysis of *Push*: The insertion function begins at a leaf of a complete binary tree and moves up toward the root. At each node on this path, $O(1)$ amount of work is done. Since a complete binary tree with n elements has a height $\lceil \log_2(n+1) \rceil$, the while loop of the insertion function is iterated $O(\log n)$ times. Hence, the complexity of *Push* is $O(\log n)$, where n is the number of elements in the heap. \square

5.6.4 Deletion from a Max Heap

When an element is to be deleted from a max heap, it is taken from the root of the heap. For instance, a deletion from the heap of Figure 5.26(d) results in the removal of the element 21. Since the resulting heap has only five elements in it, the binary tree of Figure 5.26(d) needs to be restructured to correspond to a complete binary tree with five elements. To do this, we remove the element in position 6 (i.e., the element 2). Now we have the right structure (Figure 5.27(a)), but the root is vacant and the element 2 is not in the heap. If the 2 is inserted into the root, the resulting binary tree is not a max heap. The element at the root should

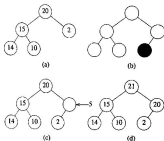


Figure 5.26: Insertion into a max heap

be the largest from among the 2 and the elements in the left and right children of the root. This element is 20. It is moved into the root, thereby creating a vacancy in position 3. Since this position has no children, the 2 may be inserted here. The resulting heap is shown in Figure 5.26(a).

Now, suppose we wish to perform another deletion. The 20 is to be deleted. Following the deletion, the heap has the binary tree structure shown in Figure 5.27(b). To get this structure, the 10 is removed from position 3. It cannot be inserted into the root, as it is not large enough. The 15 moves to the root, and we attempt to insert the 10 into position 2. This is, however, smaller than the 14 below it. So, the 14 is moved up and the 10 inserted into position 4. The resulting heap is shown in Figure 5.27(c).

Program 5.17 implements this *trickle down* strategy to delete from a heap.

Analysis of Pop: Since the height of a heap with n elements is $\lceil \log_2(n+1) \rceil$, the while loop of Program 5.17 is iterated $O(\log n)$ times. Each iteration of this loop takes $O(1)$ time. Hence, the complexity of *Pop* is $O(\log n)$. \square

```

template <class T>
void MaxHeap<T>::Push(const T& e)
// Insert e into the max heap.
    if (heapSize == capacity) { // double the capacity
        ChangeSize 1D(heap, capacity, 2*capacity);
        capacity *= 2;
    }
    int currentNode = ++heapSize;
    while (currentNode != 1 && heap[currentNode/2] < e)
    // bubble up
        heap[currentNode] = heap[currentNode/2]; // move parent down
        currentNode /= 2;
    }
    heap[currentNode] = e;
}

```

Program 5.16: Insertion into a max heap



Figure 5.27: Deletion from a heap

EXERCISES

1. Compare the run-time performance of max heaps with that of unordered and ordered linear lists as a representation for priority queues. For this comparison, program the max heap push and pop algorithms, as well as algorithms to perform these tasks on unordered and ordered linear lists that are maintained as sequential lists in a one-dimensional array. Generate a

```

template <class T>
void MaxHeap<T>::Pop()
// Delete max element.
    if (!IsEmpty()) throw "Heap is empty. Cannot delete.";
    heap[1] = T(); // delete max element

    // remove last element from heap
    T lastE = heap[heapSize - 1];

    // trickle down
    int currentNode = 1; // root
    int child = 2; // a child of currentNode
    while (child <= heapSize)
    {
        // set child to larger child of currentNode
        if (child < heapSize && heap[child] < heap[child + 1]) child++;

        // can we put lastE in currentNode?
        if (lastE >= heap[child]) break; // yes

        // no
        heap[currentNode] = heap[child]; // move child up
        currentNode = child; child *= 2; // move down a level
    }
    heap[currentNode] = lastE;
}

```

Program 5.17: Deletion from a max heap

random sequence of n values and insert these into the priority queue. Next, perform a random sequence of m inserts and deletes starting with the initial queue of n values. This sequence is to be generated so that the next operation in the sequence has an equal chance of being either an insert or a delete. Care should be taken so that the sequence does not cause the priority queue to become empty at any time. Measure the time taken for the sequence of m operations using both a max heap and an unordered list. Divide the total time by m and plot the times as a function of n . Do this for $n = 100, 500, 1000, 2000, 3000$, and 4000 . Set m to be 1000. Make some qualitative statements about the relative performance of the two representations for a max priority queue.

2. Write a C++ abstract class similar to ADT 5.2 for the ADT *MinPQ*, which defines a min priority queue. Now write a C++ class *MinHeap* that derives from this abstract class and implements all the virtual functions of *MinPQ*. The complexity of each function should be the same as that for the corresponding function of *MaxHeap*.
3. The worst-case number of comparisons performed during an insertion into a max heap can be reduced to $O(\log \log n)$ by performing a binary search on the path from the new leaf to the root. This does not affect the number of data moves though. Write an insertion algorithm that uses this strategy. Redo Exercise 1 using this insertion algorithm. Based on your experiments, what can you say about the value of this strategy over the one used in Program 5.16?

5.7 BINARY SEARCH TREES

5.7.1 Definition

A dictionary is a collection of pairs, each pair has a key and an associated element. Although naturally occurring dictionaries have several pairs that have the same key, we make the assumption here that no two pairs have the same key. The data structure, binary search tree, that we study in this section is easily extended to accommodate dictionaries in which several pairs have the same key. ADT 5.3 gives the specification of a dictionary. The data type for the keys and elements are, respectively, K and E .

```
template <class K, class E>
class Dictionary {
public:
    virtual bool IsEmpty() const = 0;
        // return true iff the dictionary is empty
    virtual pair<K, E>* Get(const K&) const = 0;
        // return pointer to the pair with specified key; return 0 if no such pair
    virtual void Insert(const pair<K, E>&) = 0;
        // insert the given pair; if key is a duplicate update associated element
    virtual void Delete(const K&) = 0;
        // delete pair with specified key
};
```

ADT 5.3: A dictionary

A binary search tree has a better performance than any of the data structures studied so far when the functions to be performed are search (i.e., *get*), insert, and delete. In fact, with a binary search tree, these functions can be performed both by key value and by rank (i.e., find an element with key k ; find the k th smallest element; delete the element with key k ; delete the fifth smallest element; insert an element and determine its rank; and so on).

Definition: A binary search tree is a binary tree. It may be empty. If it is not empty then it satisfies the following properties:

- (1) Every element has a key and no two elements have the same key (i.e., the keys are distinct).
- (2) The keys (if any) in the left subtree are smaller than the key in the root.
- (3) The keys (if any) in the right subtree are larger than the key in the root.
- (4) The left and right subtrees are also binary search trees. \square

There is some redundancy in this definition. Properties (2), (3), and (4) together imply that the keys must be distinct. So, property (1) can be replaced by the property: The root has a key.

Some examples of binary trees in which the elements have distinct keys are shown in Figure 5.28. In this figure, only the key component of each dictionary pair is shown. The tree of Figure 5.28(a) is not a binary search tree, despite the fact that it satisfies properties (1), (2), and (3). The right subtree fails to satisfy property (4). This subtree is not a binary search tree, as its right subtree has a key value (22) that is smaller than that in the subtree's root (25). The binary trees of Figures 5.28(b) and (c) are binary search trees.



Figure 5.28: Binary trees.

5.7.2 Searching a Binary Search Tree

Since the definition of a binary search tree is recursive, it is easiest to describe a recursive search method. Suppose we wish to search for an element with key k . We begin at the root. If the root is 0, then the search tree contains no elements and the search is unsuccessful. Otherwise, we compare k with the key in the root. If k is less than the key in the root, then only the left subtree is to be searched. If k is larger than the key in the root, only the right subtree needs to be searched. Otherwise, k equals the key in the root and the search terminates successfully. The subtrees may be searched recursively as in Program 5.18. This function assumes a linked representation for the search tree, which is assumed to be an object of the class *BST*. *BST* derives from the class *Tree* $\langle K, E \rangle$. The recursion of Program 5.18 is easily replaced by a while loop, as in Program 5.19.

```
template <class K, class E> // Driver
pair <K, E>* BST <K, E>::Get(const K& k)
{ // Search the binary search tree (*this) for a pair with key k.
  // If such a pair is found, return a pointer to this pair; otherwise, return 0.
    return Get(root, k);
}

template <class K, class E> // Workhorse
pair <K, E>* BST <K, E>::Get(TreeNode <pair <K, E>>* p, const K& k)
{
    if (!p) return 0;
    if (k < p->data.first) return Get(p->leftChild, k);
    if (k > p->data.first) return Get(p->rightChild, k);
    return &p->data;
}

```

Program 5.18: Recursive search of a binary search tree

We define the *rank* of a node to be its position in inorder; the first node visited in inorder has rank 1. If we wish to search by rank, each node should have an additional field *leftSize*, which is one plus the number of elements in the left subtree of the node. For the search tree of Figure 5.28(b), the nodes with keys 2, 5, 30, and 40, respectively, have *leftSize* equal to 1, 2, 3, and 1. Program 5.20 searches for the r th smallest element. To reduce code clutter, we assume that *TreeNode* has an additional field called *leftSize*. As can be seen, a binary search tree of height h can be searched by key as well as by rank in $O(h)$ time.

```

template <class K, class E> // Iterative version
pair<K, E>* BST<K, E>::Get(const K& k)
{
    TreeNode<pair<K, E>> *currentNode = root;
    while (currentNode)
        if (k < currentNode->data.first)
            currentNode = currentNode->leftChild;
        else if (k > currentNode->data.first)
            currentNode = currentNode->rightChild;
        else return &currentNode->data;
    }

    // no matching pair
    return 0;
}

```

Program 5.19: Iterative search of a binary search tree

```

template <class K, class E> // search by rank
pair<K, E>* BST<K, E>::RankGet(int r)
{ // Search the binary search tree for the rth smallest pair.
    TreeNode<pair<K, E>> *currentNode = root;
    while (currentNode)
        if (r < currentNode->leftSize) currentNode = currentNode->leftChild;
        else if (r > currentNode->rightSize)
        {
            r -= currentNode->leftSize;
            currentNode = currentNode->rightChild;
        }
        else return &currentNode->data;
    return 0;
}

```

Program 5.20: Searching a binary search tree by rank

5.7.3 Insertion into a Binary Search Tree

To insert a pair (k, e) , we must first verify that its key is different from those of existing elements. To do this, a search is carried out. If the search is unsuccessful, then the element is inserted at the point the search terminated. For instance, to insert an element with key 80 into the tree of Figure 5.28(b), we first search for 80. This search terminates unsuccessfully, and the last node examined is the one with key 40. The new pair is inserted as the right child of this node. The resulting search tree is shown in Figure 5.29(a). Figure 5.29(b) shows the result of inserting a pair with the key 35 into the search tree of Figure 5.29(a). When the dictionary already contains a pair with key k , we simply update the element associated with this key to e .



Figure 5.29: Inserting into a binary search tree

Program 5.21 implements the insert strategy just described. If a node has a *leftSize* field, then this is to be updated too. Regardless, the insertion can be performed in $O(h)$ time, where h is the height of the search tree.

5.7.4 Deletion from a Binary Search Tree

Deletion of a leaf element is quite easy. For example, to delete 35 from the tree of Figure 5.29(b), the left-child field of its parent is set to 0 and the node disposed. This gives us the tree of Figure 5.29(a). To delete the 80 from this tree, the right-child field of 40 is set to 0, obtaining the tree of Figure 5.28(b), and the node containing 80 is disposed.

The deletion of a nonleaf element that has only one child is also easy. The node containing the element to be deleted is disposed, and the single-child takes the place of the disposed node. So, to delete the element 5 from the tree of

```

template <class K, class E>
void BST<K, E>::insert(const pair<K, E>& thePair)
// Insert thePair into the binary search tree.
    // search for thePair.first, pp is parent of p
    TreeNode<K, E> *p = root, *pp = 0;
    while (p) {
        pp = p;
        if (thePair.first < p->data.first) p = p->leftChild;
        else if (thePair.first > p->data.first) p = p->rightChild;
        else // duplicate, update associated element
            (p->data.second = thePair.second); return;
    }

// perform insertion
p = new TreeNode<K, E>>(thePair);
if (root) // tree not empty
    if (thePair.first < pp->data.first) pp->leftChild = p;
    else pp->rightChild = p;
else root = p;
}

```

Program 5.21: Insertion into a binary search tree

Figure 5.29(a), we simply change the pointer from the parent node (i.e., the node containing 30) to the single-child node (i.e., the node containing 2).

When the element to be deleted is in a nonleaf node that has two children, the element is replaced by either the largest element in its left subtree or the smallest one in its right subtree. Then we proceed to delete this replacing element from the subtree from which it was taken. For instance, if we wish to delete the element with key 30 from the tree of Figure 5.29(a), then we replace it by either the largest element, 5, in its left subtree or the smallest element, 40, in its right subtree. Suppose we opt for the largest element in the left subtree. The 5 is moved into the root, and the tree of Figure 5.30(a) is obtained. Now we must delete the second 5. Since this node has only one child, the pointer from its parent is changed to point to this child. The tree of Figure 5.30(b) is obtained. One may verify that regardless of whether the replacing element is the largest in the left subtree or the smallest in the right subtree, it is originally in a node with a degree of at most one. So, deleting it from this node is quite easy. We leave the writing of the deletion function as an exercise. It should be evident that a deletion can be performed in $O(h)$ time if the search tree has a height of h .

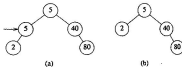


Figure 5.30: Deletion from a binary search tree

5.7.5 Joining and Splitting Binary Trees

Although search, insert, and delete are the operations most frequently performed on a binary search tree, the following additional operations are useful in certain applications:

- (a) *ThreeWayJoin*(*small*, *mid*, *big*): This creates a binary search tree consisting of the pairs initially in the binary search trees *small* and *big*, as well as the pair *mid*. It is assumed that each key in *small* is smaller than *mid.first* and that each key in *big* is greater than *mid.first*. Following the join, both *small* and *big* are empty.
- (b) *TwoWayJoin*(*small*, *big*): This joins the two binary search trees *small* and *big* to obtain a single binary search tree that contains all the pairs originally in *small* and *big*. It is assumed that all keys of *small* are smaller than all keys of *big* and that following the join both *small* and *big* are empty.
- (c) *Split*(*k*, *small*, *mid*, *big*): The binary search tree **this* is split into three parts: *small* is a binary search tree that contains all pairs of **this* that have key less than *k*; if **this* contains a pair with key *k*, then this pair is returned in the reference parameter *mid*; *big* is a binary search tree that contains all pairs of **this* that have key larger than *k*. Following the split operation **this* is empty.

A three-way join operation is particularly easy to perform. We simply obtain a new node and set its data field to *mid*, its left-child pointer to *small.root*, and its right-child pointer to *big.root*. This new node is made the root of **this*. The time taken for this operation is $O(1)$, and the height of the new tree is $\max\{\text{height}(\text{small}), \text{height}(\text{big})\} + 1$.

Consider the two-way join operation. If either *small* or *big* is empty, the result is the other tree. When neither is empty, we may first delete from *small* the pair *mid* with the largest key. Let the resulting binary search tree be *small'*. To complete the operation, we perform the three-way join operation *ThreeWayJoin*(*small'*, *mid*, *big*). The overall time required to perform the two-way join operation is $O(\text{height}(\text{small}))$, and the height of the resulting tree is $\max(\text{height}(\text{small}'), \text{height}(\text{big})) + 1$. The run time can be made $O(\min(\text{height}(\text{small}), \text{height}(\text{big})))$ if we retain with each tree its height. Then we delete the pair with the largest key from *small* if the height of *small* is no more than that of *big*; otherwise, we delete from *big* the pair with the smallest key. This is followed by a three-way join operation.

To perform a split, we first make the following observation about splitting at the root (i.e., when $k = \text{root} \rightarrow \text{data} \rightarrow \text{first}$). In this case, *small* is the left subtree of **this*, *mid* is the pair in the root, and *big* is the right subtree of **this*. If k is smaller than the key at the root, then the root together with its right subtree is to be in *big*. When k is larger than the key at the root, the root together with its left subtree is to be in *small*. Using these observations, we can perform a split by moving down the search tree **this* searching for a pair with key k . As we move down, we construct the two search trees *small* and *big*. The function to split **this* given in Program 5.22. To simplify the code, we begin with two header nodes *sHead* and *bHead* for *small* and *big*, respectively. *small* is grown as the right subtree of *sHead*; *big* is grown as the left subtree of *bHead*. $s(b)$ points to the node of *sHead* (*bHead*) at which further subtrees of **this* that are to be part of *small* (*big*) may be attached. Attaching a subtree to *small* (*big*) is done as the right (left) child of $s(b)$.

Analysis of Split: The while loop maintains the invariant that all keys in the subtree with root *currentNode* are larger than those in the tree rooted at *sHead* and smaller than those in the tree rooted at *bHead*. The correctness of the function is easy to establish, and its complexity is seen to be $O(\text{height}(*\text{this}))$. One may verify that neither *small* nor *big* has a height larger than that of **this*. \square

5.7.6 Height of a Binary Search Tree

Unless care is taken, the height of a binary search tree with n elements can become as large as n . This is the case, for instance, when Program 5.21 is used to insert the keys $\{1, 2, 3, \dots, n\}$, in this order, into an initially empty binary search tree. It can, however, be shown that when insertions and deletions are made at random using the functions given here, the height of the binary search tree is $O(\log n)$ on the average.

Search trees with a worst-case height of $O(\log n)$ are called *balanced search trees*. Balanced search trees that permit searches, inserts, and deletes to

```

template <class K, class E>
void BST<K, E>::Split(const K& k, BST<K, E>& small,
                    pair<K, E>& mid, BST<K, E>& big)
// Split the binary search tree with respect to key k.
// If (!root) {small.root = big.root = 0; return;} // empty tree
// create header nodes for small and big
TreeNode <pair<K, E>> *sHead = new TreeNode <pair<K, E>>,
    *s = sHead,
    *bHead = new TreeNode <pair<K, E>>,
    *b = bHead,
    *currentNode = root;

while (currentNode)
    if (k < currentNode->data.first) // add to big
        b->leftChild = currentNode;
        b = currentNode; currentNode = currentNode->leftChild;
    }
    else if (k > currentNode->data.first) // add to small
        s->rightChild = currentNode;
        s = currentNode; currentNode = currentNode->rightChild;
    }
    else { // split at currentNode
        s->rightChild = currentNode->leftChild;
        b->leftChild = currentNode->rightChild;
        small.root = sHead->rightChild; delete sHead;
        big.root = bHead->leftChild; delete bHead;
        mid = new pair<K, E>(currentNode->data.first,
                           currentNode->data.second);
        delete currentNode;
        return;
    }
// no pair with key k
s->rightChild = b->leftChild = 0;
small.root = sHead->rightChild; delete sHead;
big.root = bHead->leftChild; delete bHead;
mid = 0;
return;
}

```

Program 5.22 Splitting a binary search tree

be performed in $O(h)$ time exist. Most notable among these are AVL, red/black, 2-3, 2-3-4, B, and B⁺ trees. These are discussed in Chapters 10 and 11.

EXERCISES

1. Write a C++ function to delete the pair with key k from a binary search tree. What is the time complexity of your function?
2. Write a program to start with an initially empty binary search tree and make n random insertions. Use a uniform random number generator to obtain the values to be inserted. Measure the height of the resulting binary search tree and divide this height by $\log_2 n$. Do this for $n = 100, 500, 1000, 2000, 3000, \dots, 10,000$. Plot the ratio $height/\log_2 n$ as a function of n . The ratio should be approximately constant (around 2). Verify that this is so.
3. Suppose that each node in a binary search tree also has the field *leftSize* as described in the text. Write a function to insert a pair into such a binary search tree. The complexity of your function should be $O(h)$, where h is the height of the search tree. Show that this is the case.
4. Do Exercise 3, but this time write a function to delete the pair with the k th smallest key in the binary search tree.
5. Write a C++ function that implements the three-way join operation in $O(1)$ time.
6. Write a C++ function that implements the two-way join operation in $O(h)$ time, where h is the height of one of the two trees being joined.
7. Any algorithm that merges together two sorted lists of size n and m , respectively, must make at least $n + m - 1$ comparisons in the worst case. What implications does this result have on the time complexity of any comparison-based algorithm that combines two binary search trees that have n and m pairs, respectively?
8. In Chapter 7, we shall see that every comparison-based algorithm to sort n elements must make $O(n \log n)$ comparisons in the worst case. What implications does this result have on the complexity of initializing a binary search tree with n pairs?
9. Notice that a binary search tree can be used to implement a priority queue.
 - (a) Write a C++ class definition for a max priority queue that represents the priority queue as a binary search tree. This class should be a publicly derived class of *MaxPQ*.
 - (b) Write a C++ function for all virtual functions of *MaxPQ*. Your codes for *Top*, *Pop* and *Push* should have complexity $O(h)$, where h is the height of the search tree. Since h is $O(\log n)$ on average, we can

perform each these priority queue operations in average time $O(\log n)$.

- (c) Compare the actual performance of heaps and binary search trees as data structures for priority queues. For this comparison, generate random sequences of delete max and insert operations and measure the total time taken for each sequence by each of these data structures.

5.8 SELECTION TREES

5.8.1 Introduction

Suppose we have k ordered sequences, called *runs*, that are to be merged into a single ordered sequence. Each run consists of some records and is in nondecreasing order of a designated field called the *key*. Let n be the number of records in all k runs together. The merging task can be accomplished by repeatedly outputting the record with the smallest key. The smallest has to be found from k possibilities, and it could be the leading record in any of the k runs. The most direct way to merge k runs is to make $k - 1$ comparisons to determine the next record to output. For $k > 2$, we can achieve a reduction in the number of comparisons needed to find the next smallest element by using the *selection tree* data structure. There are two kinds of selection trees: *winner trees* and *loser trees*.

5.8.2 Winner Trees

A *winner tree* is a complete binary tree in which each node represents the smaller of its two children. Thus, the root node represents the smallest node in the tree. Figure 5.31 illustrates a winner tree for the case $k = 8$.

The construction of this winner tree may be compared to the playing of a tournament in which the winner is the record with the smaller key. Then, each nonleaf node in the tree represents the winner of a tournament, and the root node represents the overall winner, or the smallest key. Each leaf node represents the first record in the corresponding run. Since the records being merged are generally large, each node will contain only a pointer to the record it represents. Thus, the root node contains a pointer to the first record in run 4.

A winner tree may be represented using the sequential allocation scheme for binary trees that results from Lemma 5.4. The number above each node in Figure 5.31 is the address of the node in this sequential representation. The record pointed to by the root has the smallest key and so may be output. Now,

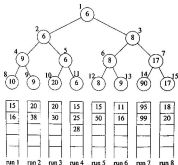


Figure 5.31: Winner tree for $k = 8$, showing the first three keys in each of the eight runs

the next record from run 4 enters the winner tree. It has a key value of 15. To restructure the tree, the tournament has to be replayed only along the path from node 11 to the root. Thus, the winner from nodes 10 and 11 is again node 11 ($15 < 20$). The winner from nodes 4 and 5 is node 4 ($9 < 15$). The winner from 3 and 5 is node 3 ($8 < 9$). The new tree is shown in Figure 5.32. The tournament is played between sibling nodes and the result put in the parent node. Lemma 5.4 may be used to compute the address of sibling and parent nodes efficiently. Each new comparison takes place at the next higher level in the tree.

Analysis of merging runs using winner trees: The number of levels in the tree is $\lceil \log_2(k + 1) \rceil$. So, the time to restructure the tree is $O(\log_2 k)$. The tree has to be restructured each time a record is merged into the output file. Hence, the time

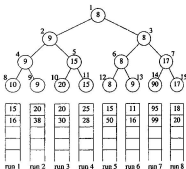


Figure 5.32: Winner tree of Figure 5.31 after one record has been output and the tree restructured (nodes that were changed are shaded)

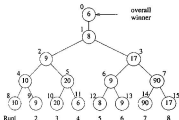
required to merge all n records is $O(n \log_2 k)$. The time required to set up the selection tree the first time is $O(k)$. Thus, the total time needed to merge the k runs is $O(n \log_2 k)$. \square

5.8.3 Loser Trees

After the record with the smallest key value is output, the winner tree of Figure 5.31 is to be restructured. Since the record with the smallest key value is in run 4, this restructuring involves inserting the next record from this run into the tree. The next record has key value 15. Tournaments are played between sibling nodes along the path from node 11 to the root. Since these sibling nodes

300 Trees

represent the losers of tournaments played earlier, we can simplify the restructuring process by placing in each nonleaf node a pointer to the record that loses the tournament rather than to the winner of the tournament. A selection tree in which each nonleaf node retains a pointer to the loser is called a *loser tree*. Figure 5.33 shows the loser tree that corresponds to the winner tree of Figure 5.31. For convenience, each node contains the key value of a record rather than a pointer to the record represented. The leaf nodes represent the first record in each run. An additional node, node 0, has been added to represent the overall winner of the tournament. Following the output of the overall winner, the tree is restructured by playing tournaments along the path from node 11 to node 1. The records with which these tournaments are to be played are readily available from the parent nodes. As a result, sibling nodes along the path from 11 to 1 are not accessed.



EXERCISES

1. Write C++ class definitions for winner and loser trees.
2. Write a function to construct a winner tree for k records. Assume that k is a power of 2. Each node at which a tournament is played should store only a pointer to the winner. Show that this construction can be carried out in time $O(k)$.
3. Do Exercise 2 for the case when k is not restricted to being a power of 2.
4. Write a function to construct a loser tree for k records. Use position 0 of your loser-tree array to store a pointer to the overall winner. Show that this construction can be carried out in time $O(k)$. Assume that k is a power of 2.
5. Do Exercise 4 for the case when k is not restricted to being a power of 2.
6. Write a function, using a tree of losers, to carry out a k -way merge of k runs, $k \geq 2$. Assume the existence of a function to initialize a loser tree in linear time. Show that if there are $n > k$ records in all k runs together, then the computing time is $O(n \log_2 k)$.
7. Do the previous exercise for the case in which a tree of winners is used. Assume the existence of a function to initialize a winner tree in linear time.
8. Compare the performance of your functions for the preceding two exercises for the case $k = 8$. Generate eight runs of data, each having 100 records. Use a random number generator for this (the keys obtained from the random number generator will need to be sorted before the merge can begin). Measure and compare the time taken to merge the eight runs using the two strategies.

5.9 FORESTS

Definition: A *forest* is a set of $n \geq 0$ disjoint trees. \square

A three-tree forest is shown in Figure 5.34. The concept of a forest is very close to that of a tree because if we remove the root of a tree, we obtain a forest. For example, removing the root of any binary tree produces a forest of two trees. In this section, we briefly consider several forest operations, including transforming a forest into a binary tree and forest traversals. In the next section, we use forests to represent disjoint sets.

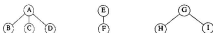


Figure 5.34: Three-tree forest

5.3.1 Transforming a Forest into a Binary Tree

To transform a forest into a single binary tree, we first obtain the binary tree representation of each of the trees in the forest and then link these binary trees together through the `rightChild` field of the root nodes. Using this transformation, the forest of Figure 5.34 becomes the binary tree of Figure 5.35.

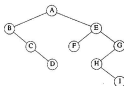


Figure 5.35: Binary tree representation of forest of Figure 5.34

We can define this transformation in a formal way as follows:

Definition: If T_1, \dots, T_n is a forest of trees, then the binary tree corresponding to this forest, denoted by $B(T_1, \dots, T_n)$,

- (1) is empty if $n = 0$
- (2) has root equal to $\text{root}(T_1)$; has left subtree equal to $B(T_1, T_2, \dots, T_n)$.

where T_{11}, \dots, T_{1m} are the subtrees of $\text{root}(T_1)$; and has right subtree $B(T_2, \dots, T_m)$. \square

5.9.2 Forest Traversals

Preorder and inorder traversals of the corresponding binary tree T of a forest F have a natural correspondence to traversals on F . Preorder traversal of T is equivalent to visiting the nodes of F in *forest preorder*, which is defined as follows:

- (1) If F is empty then return.
- (2) Visit the root of the first tree of F .
- (3) Traverse the subtrees of the first tree in forest preorder.
- (4) Traverse the remaining trees of F in forest preorder.

Inorder traversal of T is equivalent to visiting the nodes of F in *forest inorder*, which is defined as follows:

- (1) If F is empty then return.
- (2) Traverse the subtrees of the first tree in forest inorder.
- (3) Visit the root of the first tree.
- (4) Traverse the remaining trees in forest inorder.

The proofs that preorder and inorder traversals on the corresponding binary tree are the same as preorder and inorder traversals on the forest are left as exercises. There is no natural analog for postorder traversal of the corresponding binary tree of a forest. Nevertheless, we can define the *postorder traversal of a forest* as follows:

- (1) If F is empty then return.
- (2) Traverse the subtrees of the first tree of F in forest postorder.
- (3) Traverse the remaining trees of F in forest postorder.
- (4) Visit the root of the first tree of F .

In a *level-order traversal of a forest*, nodes are visited by level, beginning with the roots of each tree in the forest. Within each level, nodes are visited from left to right. One may verify that the level-order traversal of a forest and that of its associated binary tree do not necessarily yield the same result.

EXERCISES

1. Write a C++ template class definition for a forest of trees.
2. Define the inverse transformation of the one that creates the associated binary tree from a forest. Are these transformations unique?
3. Prove that the preorder traversal of a forest and the preorder traversal of its associated binary tree give the same result.
4. Prove that the inorder traversal of a forest and the inorder traversal of its associated binary tree give the same result.
5. Prove that the postorder traversal of a forest and that of its corresponding binary tree do not necessarily yield the same result.
6. Prove that the level-order traversal of a forest and that of its corresponding binary tree do not necessarily yield the same result.
7. Write a recursive function to traverse the associated binary tree of a forest in forest postorder. What are the time and space complexities of your function?
8. Do the preceding exercise for the case of forest level-order traversal.

5.10 REPRESENTATION OF DISJOINT SETS

5.10.1 Introduction

In this section we study the use of trees in the representation of sets. We shall assume that the elements of the sets are the numbers $0, 1, 2, 3, \dots, n-1$. These numbers might, in practice, be indices into a symbol table where the actual names of the elements are stored. We shall assume that the sets being represented are pairwise disjoint (i.e., if S_i and S_j , $i \neq j$, are two sets, then there is no element that is in both S_i and S_j). For example, when $n = 10$, the elements may be partitioned into three disjoint sets, $S_1 = \{0, 6, 7, 8\}$, $S_2 = \{1, 4, 9\}$, and $S_3 = \{2, 3, 5\}$. Figure 5.36 shows one possible representation for these sets. In this representation, each set is represented as a tree. Notice that for each set we have linked the nodes from the children to the parent, rather than our usual method of linking from the parent to the children. The reason for this change in linkage will become apparent when we discuss the implementation of set operations.

The operations we wish to perform on these sets are:

- (1) *Disjoint set union*. If S_i and S_j are two disjoint sets, then their union $S_i \cup S_j = \{\text{all elements } x \text{ such that } x \text{ is in } S_i \text{ or } S_j\}$. Thus, $S_1 \cup S_2 = \{0, 6, 7, 8, 1, 4, 9\}$. Since we have assumed that all sets are

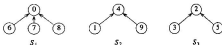


Figure 5.36: Possible tree representation of sets

disjoint, we can assume that following the union of S_i and S_j , the sets S_i and S_j do not exist independently; that is, they are replaced by $S_i \cup S_j$ in the collection of sets.

- (2) *Find(i)*. Find the set containing element i . Thus, 3 is in set S_3 , and 8 is in set S_1 .

5.10.2 Union and Find Operations

Let us consider the union operation first. Suppose that we wish to obtain the union of S_1 and S_2 (see Figure 5.36). Since we have linked the nodes from children to parent, we simply make one of the trees a *subtree* of the other. $S_1 \cup S_2$ could then have one of the representations of Figure 5.37.



Figure 5.37: Possible representations of $S_1 \cup S_2$

To obtain the union of two sets, all that has to be done is to set the parent field of one of the roots to the other root. This can be accomplished easily if, with each set name, we keep a pointer to the root of the tree representing that set.

306 Trees

If, in addition, each root has a pointer to the set name, then to determine which set an element is currently in, we follow parent links to the root of its tree and use the pointer to the set name. The data representation for S_1 , S_2 , and S_3 may then take the form shown in Figure 5.38.

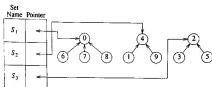


Figure 5.38: Data representation for S_1 , S_2 , and S_3 .

In presenting the union and find algorithms we shall ignore the actual set names and just identify sets by the roots of the trees representing them. This will simplify the discussion. The transition to set names is easy. If we determine that element i is in a tree with root j , and j has a pointer to entry k in the set name table, then the set name is just `name[k]`. If we wish to unite sets S_i and S_j , then we wish to unite the trees with roots `FindPointer(S_i)` and `FindPointer(S_j)`. Here `FindPointer` is a function that takes a set name and determines the root of the tree that represents it. This is done by an examination of the [set name, pointer] table. As we shall see, in many applications the set name is just the element at the root. The operation of `Find()` now becomes: Determine the root of the tree containing element i . The function `Union(i, j)` requires us to join the trees with roots i and j . Another simplifying assumption we shall make is that the set elements are the numbers 0 through $n-1$.

Since the set elements are numbered 0 through $n-1$, we represent the tree nodes using an array `parent[]`. The i th element of this array represents the tree node that contains element i . This array element gives the parent pointer of the corresponding tree node. Figure 5.39 shows this representation of the sets, S_1 , S_2 , and S_3 of Figure 5.36. Notice that root nodes have a parent of -1 .

Program 5.23 contains the class definition and constructor for the data structure.

i	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
parent	-1	4	-1	2	-1	2	0	0	0	4

Figure 5.39: Array representation of S_1 , S_2 , and S_3 of Figure 5.36

```

class Sets {
public:
    // set operations follow
    .
    .
private:
    int *parent;
    int n; // number of set elements
};

Sets::Sets(int numberOfElements)
{
    if (numberOfElements < 2) throw "Must have at least 2 elements.";
    n = numberOfElements;
    parent = new int[n];
    fill(parent, parent + n, -1);
}

```

Program 5.23: Class definition and constructor for Sets

We can now implement *Find(i)* by simply following the indices starting at i and continuing until we reach a node with parent value -1 . For example, *Find(5)* starts at 5 and then moves to 5's parent, 2. Since *parent*[2] = -1 , we have reached the root. The operation *Union(i, j)* is equally simple. We pass in two trees with roots i and j , $i \neq j$. Assuming that we adopt the convention that the first tree becomes a subtree of the second, the statement *parent*[i] = j accomplishes the union. Program 5.24 implements the union and find operations as just discussed.

Analysis of SimpleUnion and SimpleFind: Although these two functions are very easy to state, their performance characteristics are not very good. For

```

void Set::SimpleUnion(int i, int j)
// Replace the disjoint sets with roots i and j, i >= j with their union.
    parent[i] = j;
}

int Set::SimpleFind(int i)
// Find the root of the tree containing element i.
    while (parent[i] >= 0) i = parent[i];
    return i;
}

```

Program 5.24: Simple functions for union and find

instance, if we start off with n elements each in a set of its own (i.e., $S_i = \{i\}$, $0 \leq i < n$), then the initial configuration consists of a forest with n nodes, and $\text{parent}[i] = -1$, $0 \leq i < n$. Now let us process the following sequence of operations:

$\text{Union}(0,1), \text{Union}(1,2), \text{Union}(2,3), \text{Union}(3,4), \dots, \text{Union}(n-2,n-1)$
 $\text{Find}(0), \text{Find}(1), \dots, \text{Find}(n-1)$

This sequence results in the degenerate tree of Figure 5.40. Since the time taken for a union is constant, the $n-1$ unions can be processed in time $O(n)$. However, each find operation requires following a sequence of parent pointers from the element to be found to the root. Since the time required to process a find for an element at level i of a tree is $O(i)$, the total time needed to process the n finds is $O(\sum_{i=0}^{n-1} i) = O(n^2)$. \square

We can improve performance by avoiding the creation of degenerate trees. In order to accomplish this we shall make use of a weighting rule for $\text{Union}(i,j)$.

Definition [Weighting rule for $\text{Union}(i,j)$]: If the number of nodes in the tree with root i is less than the number in the tree with root j , then make j the parent of i ; otherwise make i the parent of j . \square

When we use the weighting rule to perform the sequence of set unions given before, we obtain the trees of Figure 5.41. In this figure, the unions have been modified so that the input parameter values correspond to the roots of the trees to be combined.

To implement the weighting rule, we need to know how many nodes there are in every tree. To do this easily, we maintain a *count* field in the root of every



Figure 5.40: Degenerate tree

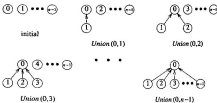


Figure 5.41: Trees obtained using the weighting rule

340 Trees

tree. If i is a root node, then $\text{count}[i]$ equals the number of nodes in that tree. Since all nodes other than the roots of trees have a non-negative number in the *parent* field, we can maintain the count in the *parent* field of the roots as a negative number. Initially, the *parent* field of all nodes contains -1 . Using this convention, we obtain the union function of Program 5.25.

```
void Set::WeightedUnion(int i, int j)
// Union sets with roots i and j, (xy), using the weighting rule.
// parent[i] = -count[i] and parent[j] = -count[j].
{
    int temp = parent[i] + parent[j];
    if (parent[i] > parent[j]) { // i has fewer nodes
        parent[i] = j;
        parent[j] = temp;
    }
    else { // j has fewer nodes (or i and j have the same number of nodes)
        parent[j] = i;
        parent[i] = temp;
    }
}
```

Program 5.25: Union function with weighting rule

Analysis of WeightedUnion and SimpleFind: The time required to perform a union has increased somewhat but is still bounded by a constant (i.e., it is $O(1)$). The find function remains unchanged. The maximum time to perform a find is determined by Lemma 5.5.

Lemma 5.5: Assume that we start with a forest of trees, each having one node. Let T be a tree with m nodes created as a result of a sequence of unions each performed using function *WeightedUnion*. The height of T is no greater than $\lceil \log_2 m \rceil + 1$.

Proof: The lemma is clearly true for $m = 1$. Assume it is true for all trees with i nodes, $i \leq m - 1$. We shall show that it is also true for $i = m$. Let T be a tree with m nodes created by function *WeightedUnion*. Consider the last union operation performed, *Union*(k, j). Let a be the number of nodes in tree j and $m - a$ the number in k . Without loss of generality we may assume $1 \leq a \leq m/2$. Then the height of T is either the same as that of k or is one more than that of j . If the former is the case, the height of T is $\leq \lceil \log_2 (m - a) \rceil + 1 \leq \lceil \log_2 m \rceil + 1$. If the

latter is the case, the height of T is $\leq \lfloor \log_2 n \rfloor + 2 \leq \lfloor \log_2 m/2 \rfloor + 2 \leq \lfloor \log_2 m \rfloor + 1$. \square

Example 5.3 shows that the bound of Lemma 5.5 is achievable for some sequence of unions.

Example 5.3: Consider the behavior of function *WeightedUnion* on the following sequence of unions starting from the initial configuration $\text{parent}[i] = -\text{count}[i] = -1$, $0 \leq i < n = 8$:

Union(0,1), *Union*(2,3), *Union*(4,5), *Union*(6,7),
Union(0,2), *Union*(4,6), *Union*(0,4)

The trees of Figure 5.42 are obtained. As is evident, the height of each tree with m nodes is $\lfloor \log_2 m \rfloor + 1$. \square

From Lemma 5.5, it follows that the time to process a find is $O(\log m)$ if there are m elements in a tree. If an intermixed sequence of $u - 1$ union and f find operations is to be processed, the time becomes $O(u + f \log u)$, as no tree has more than u nodes in it. Of course, we need $O(n)$ additional time to initialize the n -tree forest. \square

Surprisingly, further improvement is possible. This time the modification will be made in the find algorithm using the *collapsing rule*.

Definition (Collapsing rule): If j is a node on the path from i to its root and $\text{parent}[j] \neq \text{root}(j)$, then set $\text{parent}[j]$ to $\text{root}(i)$. \square

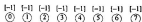
Function *CollapsingFind* (Program 5.26) incorporates the collapsing rule.

Example 5.4: Consider the tree created by function *WeightedUnion* on the sequence of unions of Example 5.3. Now process the following eight finds:

Find(7), *Find*(7), \dots , *Find*(7)

If *SimpleFind* is used, each *Find*(7) requires going up three parent link fields for a total of 24 moves to process all eight finds. When *CollapsingFind* is used, the first *Find*(7) requires going up three links and then resetting two links. Now that even though only two parent links need to be reset, function *CollapsingFind* will actually reset three (the parent of 4 is reset to 0). Each of the remaining seven finds requires going up only one link field. The total cost is now only 13 moves. \square

Analysis of *WeightedUnion* and *CollapsingFind*: Use of the collapsing rule roughly doubles the time for an individual find. However, it reduces the worst-



(a) Initial height-1 trees

(b) Height-2 trees following *Union* (0,1), (2,3), (4,5), and (6,7)(c) Height-3 trees following *Union* (0,2) and (4,6)(d) Height-4 tree following *Union* (0,4)

Figure 5.42: Trees achieving worst-case bound

```

Int Sets::CollapsingFind(int i)
// Find the root of the tree containing element i.
// Use the collapsing rule to collapse all nodes from i to the root.
    for (int r = i; parent[r] >= 0; r = parent[r]); // find root
    while (i != r) // collapse
        int s = parent[i];
        parent[i] = r;
        i = s;
    }
    return r;
}

```

Program 5.26c: Collapsing rule

case time over a sequence of finds. The worst-case complexity of processing a sequence of unions and finds using *WeightedUnion* and *CollapsingFind* is stated in Lemma 5.6. This lemma makes use of a function $\alpha(p, q)$ that is related to a functional inverse of Ackermann's function $A(i, j)$. These functions are defined as follows:

$$\begin{aligned}
 A(i, j) &= 2^j, & \text{for } j \geq 1 \\
 A(i, 1) &= A(i-1, 2) & \text{for } i \geq 2 \\
 A(i, j) &= A(i-1, A(i, j-1)) & \text{for } i, j \geq 2
 \end{aligned}$$

$$\alpha(p, q) = \min\{z \geq 1 \mid A(z, \lfloor p/q \rfloor) > \log_2 q\}, \quad p \geq q \geq 1$$

The function $A(i, j)$ is a very rapidly growing function. Consequently, it grows very slowly as p and q are increased. In fact, since $A(4, 1) = 16$, $\alpha(p, q) \leq 3$ for $q < 2^{16} = 65,536$ and $p \geq q$. Since $A(4, 1)$ is a very large number and in our application q will be the number, n , of set elements and p will be $n + f$ (f is the number of finds), $\alpha(p, q) \leq 4$ for all practical purposes. \square

Lemma 5.6 [Tarjan and Van Leeuwen]: Assume that we start with a forest of trees, each having one node. Let $T(f, u)$ be the maximum time required to process any intermixed sequence of f finds and u unions. Assume that $u \geq n/2$. Then

$$k_1(n + f \alpha(f + n, n)) \leq T(f, u) \leq k_2(n + f \alpha(f + n, n))$$

for some positive constants k_1 and k_2 . \square

314 Trees

The requirement that $n \geq n/2$ in Lemma 5.6, is really not significant, as when $n < n/2$, some elements are involved in no union operation. These elements remain in singleton sets throughout the sequence of union and find operations and can be eliminated from consideration, as find operations that involve these can be done in $O(1)$ time each. Even though the function $\alpha(f, n)$ is a very slowly growing function, the complexity of our solution to the set representation problem is not linear in the number of unions and finds. The space requirements are one node for each element.

In the exercises, we explore alternatives to the weight rule and the collapsing rule that preserve the time bounds of Lemma 5.6.

5.10.3 Application to Equivalence Classes

Consider the equivalence pairs processing problem of Section 4.9. The equivalence classes to be generated may be regarded as sets. These sets are disjoint, as no polygon can be in two equivalence classes. Initially, all n polygons are in an equivalence class of their own; thus $\text{parent}[i] = -1, 0 \leq i < n$. If an equivalence pair, $i = j$, is to be processed, we must first determine the sets containing i and j . If these are different, then the two sets are to be replaced by their union. If the two sets are the same, then nothing is to be done, as the relation $i = j$ is redundant; i and j are already in the same equivalence class. To process each equivalence pair we need to perform two finds and at most one union. Thus, if we have n polygons and m equivalence pairs, we need to spend $O(n)$ time to set up the initial n -tree forest, and then we need to process $2m$ finds and at most $\min\{n-1, m\}$ unions. (Note that after $n-1$ unions, all n polygons will be in the same equivalence class and no more unions can be performed.) If we use *WeightedUnion* and *CollapsingFind*, the total time to process the equivalence relations is $O(n + m\alpha(2m, \min\{n-1, m\}))$. Although this is slightly worse than the algorithm of Section 4.9, it needs less space and is on line. By “on line,” we mean that as each equivalence is processed, we can tell which equivalence class each polygon is in.

Example 5.8: Consider the equivalence pairs example of Chapter 4. Initially, there are 12 trees, one for each variable. $\text{parent}[i] = -1, 0 \leq i < 12$. The tree configuration following the processing of each equivalence pair is shown in Figure 5.43. Each tree represents an equivalence class. It is possible to determine if two elements are currently in the same equivalence class at each stage of the processing simply by making two finds. \square

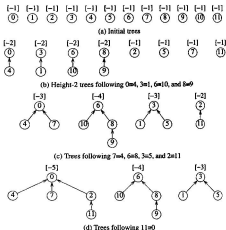


Figure 5.43: Trees for Example 5.5

EXERCISES

1. Suppose we start with n sets, each containing a distinct element.
 - (a) Show that if u unions are performed, then no set contains more than $u + 1$ elements.
 - (b) Show that at most $n - 1$ unions can be performed before the number

of sets becomes 1.

- (c) Show that if fewer than $\lceil n/2 \rceil$ unions are performed, then at least one set with a single element in it remains.
 - (d) Show that if n unions are performed, then at least $\max\{n - 2n, 0\}$ singleton sets remain.
2. Using the result of Example 5.3, draw the trees after processing the instruction `union(11,9)`.
 3. Experimentally compare the performance of *SimpleUnion* and *SimpleFind* (Program 5.24) with *WeightedUnion* (Program 5.25) and *CollapsingFind* (Program 5.26). For this, generate a random sequence of union and find operations.
 4. (a) Write a function *HeightUnion* that uses the *height rule* for union operations instead of the *weighting rule*. This rule is defined below:
Definition [Height Rule]: If the height of tree i is less than that of tree j , then make j the parent of i , otherwise make i the parent of j . \square
 Your function must run in $O(1)$ time and should maintain the height of each tree as a negative number in the *parent* field of the root.
 (b) Show that the height bound of Lemma 5.5 applies to trees constructed using the height rule.
 (c) Give an example of a sequence of unions that start with singleton sets and create trees whose height equals the upper bound given in Lemma 5.5. Assume that each union is performed using the height rule.
 (d) Experiment with functions *WeightedUnion* (Program 5.25) and *HeightUnion* to determine which one produces better results when used in conjunction with function *CollapsingFind* (Program 5.26).
 5. (a) Write a function *SplittingFind* that uses *path splitting* for the find operations instead of *path collapsing*. This is defined below:
Definition [Path Splitting]: In path splitting, the parent pointer in each node (except the root and its child) on the path from i to the root is changed to point to the node's grandparent. \square
 Note that when path splitting is used, a single pass from i to the root suffices. Tarjan and Van Leeuwen have shown that Lemma 5.6 holds when path splitting is used in conjunction with either the *weight* or *height rule* for unions.
 (b) Experiment with functions *CollapsingFind* (Program 5.26) and *SplittingFind* to determine which produces better results when used in conjunction with function *WeightedUnion* (Program 5.25).

6. (a) Write a function *HalvingFind* that uses *path halving* for the find operations instead of path collapsing. This is defined below:

Definition [Path Halving]: In path halving, the parent pointer of every other node (except the root and its child) on the path from i to the root is changed to point to the node's grandparent. \square

Note that path halving, like path splitting (Exercise 5) can be implemented with a single pass from i to the root. However, in path halving, only half as many pointers are changed as in path splitting. Tarjan and Van Leeuwen have shown that Lemma 5.6 holds when path halving is used in conjunction with either the weight or height rule for unions.

- (b) Experiment with functions *CollapsingFind* and *HalvingFind* to determine which one produces better results when used in conjunction with function *WeightedUnion*.

5.11 COUNTING BINARY TREES

As a conclusion to our chapter on trees, we consider three disparate problems that amazingly have the same solution. We wish to determine the number of distinct binary trees having n nodes, the number of distinct permutations of the numbers from 1 through n obtainable by a stack, and the number of distinct ways of multiplying $n + 1$ matrices. Let us begin with a quick look at these problems.

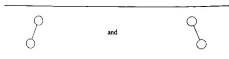
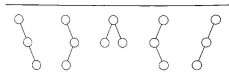
5.11.1 Distinct Binary Trees

We know that if $n = 0$ or $n = 1$, there is only one binary tree. If $n = 2$, then there are two distinct trees (Figure 5.44), and if $n = 3$, there are five such trees (Figure 5.45). How many distinct trees are there with n nodes? Before deriving a solution, we will examine the two remaining problems. You might attempt to sketch out a solution of your own before reading farther.

5.11.2 Stack Permutations

In Section 5.3, we introduced preorder, inorder, and postorder traversals and indicated that each traversal requires a stack. Suppose we have the preorder sequence $A B C D E F G H I$ and the inorder sequence $B C A E D G H F I$ of the same binary tree. Does such a pair of sequences uniquely define a binary tree? Put another way, can this pair of sequences come from more than one binary tree?

To construct the binary tree from these sequences, we look at the first letter

Figure 5.44: Distinct binary trees with $n = 2$ Figure 5.45: Distinct binary trees with $n = 3$

in the preorder sequence, A . This letter must be the root of the tree by definition of the preorder traversal (PLR). We also know by definition of the inorder traversal (ILR) that all nodes preceding A in the inorder sequence ($B\ C$) are in the left subtree, and the remaining nodes ($E\ D\ G\ H\ F\ I$) are in the right subtree. Figure 5.46(a) is our first approximation to the correct tree.

Moving right in the preorder sequence, we find B as the next root. Since no node precedes B in the inorder sequence, B has an empty left subtree, which means that C is in its right subtree. Figure 5.46(b) is the next approximation. Continuing in this way, we arrive at the binary tree of Figure 5.47(a). By formalizing this argument (see the exercises), we can verify that every binary tree has a unique pair of preorder/inorder sequences.

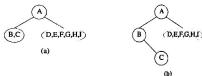


Figure 5.46: Constructing a binary tree from its inorder and preorder sequences.

Let the nodes of an n -node binary tree be numbered from 1 through n . The *inorder* permutation defined by such a binary tree is the order in which its nodes are visited during an inorder traversal of the tree. A *preorder* permutation is similarly defined.

As an example, consider the binary tree of Figure 5.47(a) with the node numbering of Figure 5.47(b). Its preorder permutation is 1, 2, ..., 9, and its inorder permutation is 2, 3, 1, 5, 4, 7, 8, 6, 9.

If the nodes of the tree are numbered such that its preorder permutation is 1, 2, ..., n , then from our earlier discussion it follows that distinct binary trees define distinct inorder permutations. Thus, the number of distinct binary trees is equal to the number of distinct inorder permutations obtainable from binary trees having the preorder permutation, 1, 2, ..., n .

Using the concepts of an inorder permutation, we can show that the number of distinct permutations obtainable by passing the numbers 1 through n through a stack and deleting in all possible ways is equal to the number of distinct binary trees with n nodes (see the exercises). If we start with the numbers 1, 2, and 3, then the possible permutations obtainable by a stack are

$$(1, 2, 3) (1, 3, 2) (2, 1, 3) (2, 3, 1) (3, 2, 1)$$

Obtaining (3, 1, 2) is impossible. Each of these five permutations corresponds to one of the five distinct binary trees with three nodes (Figure 5.48).

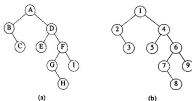


Figure 5.47: Binary tree constructed from its inorder and preorder sequences



Figure 5.48: Binary trees corresponding to five permutations

5.11.3 Matrix Multiplication

Another problem that surprisingly has a connection with the previous two involves the product of n matrices. Suppose that we wish to compute the product of n matrices:

$$M_1 * M_2 * \cdots * M_n$$

Since matrix multiplication is associative, we can perform these multiplications

in any order. We would like to know how many different ways we can perform these multiplications. For example, if $n = 3$, there are two possibilities:

$$\begin{aligned}(M_1 * M_2) * M_3 \\ M_1 * (M_2 * M_3)\end{aligned}$$

and if $n = 4$, there are five:

$$\begin{aligned}((M_1 * M_2) * M_3) * M_4 \\ (M_1 * (M_2 * M_3)) * M_4 \\ M_1 * ((M_2 * M_3) * M_4) \\ (M_1 * (M_2 * (M_3 * M_4))) \\ ((M_1 * M_2) * (M_3 * M_4))\end{aligned}$$

Let b_n be the number of different ways to compute the product of n matrices. Then $b_2 = 1$, $b_3 = 2$, and $b_4 = 5$. Let M_{ij} , $i \leq j$, be the product $M_i * M_{i+1} * \cdots * M_j$. The product we wish to compute is M_{1n} . We may compute M_{1n} by computing any one of the products $M_{1i} * M_{i+1,n}$, $1 \leq i \leq n$. The number of distinct ways to obtain M_{1i} and $M_{i+1,n}$ are b_i and b_{n-i} , respectively. Therefore, letting $b_1 = 1$, we have

$$b_n = \sum_{i=0}^{n-1} b_i b_{n-i}, \quad n \geq 1$$

If we can determine the expression for b_n only in terms of n , then we have a solution to our problem.

Now instead let b_n be the number of distinct binary trees with n nodes. Again an expression for b_n in terms of n is what we want. Then we see that b_n is the sum of all the possible binary trees formed in the following way: a root and two subtrees with b_i and b_{n-i-1} nodes, for $0 \leq i < n$ (Figure 5.49). This explanation says that

$$b_n = \sum_{i=0}^{n-1} b_i b_{n-i-1}, \quad n \geq 1, \quad \text{and } b_0 = 1 \quad (5.3)$$

This formula and the previous one are essentially the same. Therefore, the number of binary trees with n nodes, the number of permutations of 1 to n obtainable with a stack, and the number of ways to multiply $n + 1$ matrices are all equal.

5.11.4 Number of Distinct Binary Trees

To obtain the number of distinct binary trees with n nodes, we must solve the recurrence of Eq. (5.3). To begin we let

Figure 5.49: Decomposing b_n .

$$B(x) = \sum_{i \geq 0} b_i x^i \quad (5.4)$$

which is the generating function for the number of binary trees. Next observe that by the recurrence relation we get the identity

$$xB^2(x) = B(x) - 1$$

Using the formula to solve quadratics and the fact that $B(0) = b_0 = 1$ (Eq.(5.3)), we get

$$B(x) = \frac{1 - \sqrt{1-4x}}{2x}$$

We can use the binomial theorem to expand $(1-4x)^{1/2}$ to obtain

$$B(x) = \frac{1}{2x} \left(1 - \sum_{n \geq 0} \binom{1/2}{n} (-4x)^n \right) = \sum_{n \geq 0} \binom{1/2}{n+1} (-1)^n 2^{2n+1} x^n \quad (5.5)$$

Comparing Eqs. (5.4) and (5.5), we see that b_n , which is the coefficient of x^n in $B(x)$, is

$$\binom{1/2}{n+1} (-1)^n 2^{2n+1}$$

Some simplification yields the more compact form

$$b_n = \frac{1}{n+1} \left[\frac{2n}{n} \right]$$

which is approximately

$$b_n = O(4^n n^{-3/2})$$

EXERCISES

1. Prove that every binary tree is uniquely defined by its preorder and inorder sequences.
2. Do the inorder and postorder sequences of a binary tree uniquely define the binary tree? Prove your answer.
3. Do the inorder and preorder sequences of a binary tree uniquely define the binary tree? Prove your answer.
4. Do the inorder and level-order sequences of a binary tree uniquely define the binary tree? Prove your answer.
5. Write an algorithm to construct the binary tree with given preorder and inorder sequences.
6. Repeat Exercise 5 with the inorder and postorder sequences.
7. Prove that the number of distinct permutations of $1, 2, \dots, n$ obtainable by a stack is equal to the number of distinct binary trees with n nodes. (Hint: Use the concept of an inorder permutation of a tree with preorder permutation $1, 2, \dots, n$).

5.12 REFERENCES AND SELECTED READINGS

For more on trees, see *The Art of Computer Programming: Fundamental Algorithms*, Third Edition, by D. Knuth, Addison-Wesley, Reading, MA, 1998 and "Handbook of data structures and applications," edited by D. Mehu and S. Sahni, Chapman & Hall/CRC, Boca Raton, 2005.

CHAPTER 6

Graphs

6.1 THE GRAPH ABSTRACT DATA TYPE

6.1.1 Introduction

The first recorded evidence of the use of graphs dates back to 1736, when Leonhard Euler used them to solve the now classical Königsberg bridge problem. In the town of Königsberg (now Kaliningrad) the river Pregel (Pregolya) flows around the island Kneighof and then divides into two. There are, therefore, four land areas that have this river on its borders (see Figure 6.1(a)). These land areas are interconnected by seven bridges labeled *a*–*g*. The land areas themselves are labeled *A*–*D*. The Königsberg bridge problem is to determine whether, starting at one land area, it is possible to walk across all the bridges exactly once in returning to the starting land area. One possible walk is

- start from land area *B*
- walk across bridge *a* to island *A*

- take bridge e to area D
- take bridge g to C
- take bridge d to A
- take bridge b to B
- take bridge f to D

This walk does not go across all bridges exactly once, nor does it return to the starting land area B . Euler answered the Königsberg bridge problem in the negative: The people of Königsberg will not be able to walk across each bridge exactly once and return to the starting point. He solved the problem by representing the land areas as vertices and the bridges as edges in a graph (actually a multigraph) as in Figure 6.1(b). His solution is elegant and applies to all graphs. Defining the *degree* of a vertex to be the number of edges incident to it, Euler showed that there is a walk starting at any vertex, going through each edge exactly once and terminating at the start vertex iff the degree of each vertex is even. A walk that does this is called *Eulerian*. There is no Eulerian walk for the Königsberg bridge problem, as all four vertices are of odd degree.

Since this first application, graphs have been used in a wide variety of applications. Some of these applications are: analysis of electrical circuits, finding shortest routes, project planning, identification of chemical compounds, statistical mechanics, genetics, cybernetics, linguistics, social sciences, and so on. Indeed, it might well be said that of all mathematical structures, graphs are the most widely used.

6.1.2 Definitions

A graph, G , consists of two sets, V and E . V is a finite, nonempty set of vertices. E is a set of pairs of vertices; these pairs are called *edges*. $V(G)$ and $E(G)$ will represent the sets of vertices and edges, respectively, of graph G . We will also write $G = (V, E)$ to represent a graph. In an *undirected* graph the pair of vertices representing any edge is unordered. Thus, the pairs $\{u, v\}$ and $\{v, u\}$ represent the same edge. In a *directed* graph each edge is represented by a directed pair $\langle u, v \rangle$; u is the *tail* and v the *head* of the edge[†]. Therefore, $\langle u, v \rangle$ and $\langle v, u \rangle$ represent two different edges. Figure 6.2 shows three graphs: G_1 , G_2 , and G_3 . The graphs G_1 and G_2 are undirected. G_3 is a directed graph.

The set representation of each of these graphs is

[†]Often, both the undirected edge $\{i, j\}$ and the directed edge $\langle i, j \rangle$ are written as (i, j) . Which is meant is deduced from the context. In this book, we refrain from this practice.

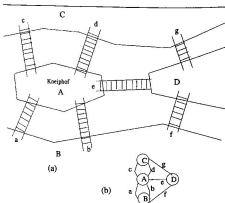


Figure 4.1: (a) Section of the river Pregel in Königsberg; (b) Euler's graph

$$\begin{aligned}
 V(G_1) &= \{0, 1, 2, 3\}; & E(G_1) &= \{(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)\} \\
 V(G_2) &= \{0, 1, 2, 3, 4, 5, 6\}; & E(G_2) &= \{(0, 1), (0, 2), (1, 3), (1, 4), (2, 5), (2, 6)\} \\
 V(G_3) &= \{0, 1, 2\}; & E(G_3) &= \{ \langle 0, 1 \rangle, \langle 1, 0 \rangle, \langle 1, 2 \rangle \}.
 \end{aligned}$$

Notice that the edges of a directed graph are drawn with an arrow from the tail to the head. The graph G_2 is a tree; the graphs G_1 and G_3 are not.

Since we define the edges and vertices of a graph as sets, we impose the following restrictions on graphs:

- (1) A graph may not have an edge from a vertex, v , back to itself. That is,



Figure 6.2: Three sample graphs

edges of the form $\langle v, v \rangle$ and $\langle v, v \rangle$ are not legal. Such edges are known as *self edges* or *self loops*. If we permit self edges, we obtain a data object referred to as a *graph with self edges*. An example is shown in Figure 6.3(a).

- (2) A graph may not have multiple occurrences of the same edge. If we remove this restriction, we obtain a data object referred to as a *multigraph* (see Figure 6.3(b)).



Figure 6.3: Examples of graphlike structures

The number of distinct unordered pairs $\{u, v\}$ with $u \neq v$ in a graph with n vertices is $n(n-1)/2$. This is the maximum number of edges in any n -vertex, undirected graph. An n -vertex, undirected graph with exactly $n(n-1)/2$ edges is said to be *complete*. The graph G_1 of Figure 6.2(a) is the complete graph on

four vertices, whereas G_1 and G_2 are not complete graphs. In the case of a directed graph on n vertices, the maximum number of edges is $n(n-1)$.

If $\langle u, v \rangle$ is an edge in $E(G)$, then we shall say the vertices u and v are *adjacent* and that the edge $\langle u, v \rangle$ is *incident* on vertices u and v . The vertices adjacent to vertex 1 in G_1 are 3, 4, and 0. The edges incident on vertex 2 in G_1 are $\langle 0, 2 \rangle$, $\langle 2, 5 \rangle$, and $\langle 2, 6 \rangle$. If $\langle u, v \rangle$ is a directed edge, then vertex u is *adjacent to* v , and v is *adjacent from* u . The edge $\langle u, v \rangle$ is incident to u and v . In G_2 , the edges incident to vertex 1 are $\langle 0, 1 \rangle$, $\langle 1, 0 \rangle$, and $\langle 1, 2 \rangle$.

A *subgraph* of G is a graph G' such that $V(G') \subseteq V(G)$ and $E(G') \subseteq E(G)$. Figure 6.4 shows some of the subgraphs of G_1 and G_2 .

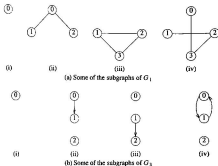


Figure 6.4: Some subgraphs

A *path* from vertex u to vertex v in graph G is a sequence of vertices $u, i_1, i_2, \dots, i_k, v$ such that $\langle u, i_1 \rangle, \langle i_1, i_2 \rangle, \dots, \langle i_k, v \rangle$ are edges in $E(G)$. If G is directed, then the path consists of $\langle u, i_1 \rangle, \langle i_1, i_2 \rangle, \dots, \langle i_k, v \rangle$ edges in $E(G)$. The *length* of a path is the number of edges on it. A *simple path* is a

path in which all vertices except possibly the first and last are distinct. A path such as $\langle 0,1 \rangle$, $\langle 1,3 \rangle$, $\langle 3,2 \rangle$, is also written as 0,1,3,2. Paths 0,1,3,2 and 0,1,3,1 of G_1 are both of length 3. The first is a simple path; the second is not. 0,1,2 is a simple directed path in G_2 . 0,1,2,1 is not a path in G_2 , as the edge $\langle 2,1 \rangle$ is not in $E(G_2)$.

A *cycle* is a simple path in which the first and last vertices are the same. 0,1,2,0 is a cycle in G_1 . 0,1,0 is a cycle in G_2 . For the case of directed graphs we normally add the prefix "directed" to the terms cycle and path.

In an undirected graph, G , two vertices u and v are said to be *connected* iff there is a path in G from u to v (since G is undirected, this means there must also be a path from v to u). An undirected graph is said to be *connected* iff for every pair of distinct vertices u and v in $V(G)$ there is a path from u to v in G . Graphs G_1 and G_2 are connected, whereas G_3 of Figure 6.5 is not. A *connected component* (or simply a *component*), H , of an undirected graph is a *maximal* connected subgraph. By *maximal*, we mean that G contains no other subgraph that is both connected and properly contains H . G_3 has two components, H_1 and H_2 (see Figure 6.5).

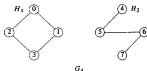


Figure 6.5: A graph with two connected components

A *tree* is a connected acyclic (i.e., has no cycles) graph.

A directed graph G is said to be *strongly connected* iff for every pair of distinct vertices u and v in $V(G)$, there is a directed path from u to v and also from v to u . The graph G_3 is not strongly connected, as there is no path from vertex 2 to 1. A *strongly connected component* is a maximal subgraph that is strongly connected. G_3 has two strongly connected components (see Figure 6.6).

The *degree* of a vertex is the number of edges incident to that vertex. The degree of vertex 0 in G_1 is 3. If G is a directed graph, we define the *in-degree* of a vertex v to be the number of edges for which v is the head. The *out-degree* is defined to be the number of edges for which v is the tail. Vertex 1 of G_2 has in-

Figure 6.6c: Strongly connected components of G_3

degree 1, out-degree 2, and degree 3. If d_i is the degree of vertex i in a graph G with n vertices and e edges, then the number of edges is

$$e = \left(\sum_{i=0}^{n-1} d_i \right) / 2$$

In the remainder of this chapter, we shall refer to a directed graph as a *digraph*. When we use the term *graph*, we assume that it is an undirected graph. Now that we have defined all the terminology we will need, let us consider the graph as an ADT. The graph data structure is specified as an abstract class in ADT 6.1. We use an abstract class, because many different representations for a graph are possible. Classes that actually implement a graph using a specific graph representation can derive publicly from the abstract class *Graph*.

The operations in ADT 6.1 are a basic set in that they allow us to create any arbitrary graph and do some elementary tests. In later sections of this chapter we will see functions that traverse a graph (depth-first or breadth-first search) and that determine if a graph has special properties (e.g., connected, biconnected, etc.).

6.1.3 Graph Representations

Although several representations for graphs are possible, we shall study only the three most commonly used: adjacency matrices, adjacency lists, and adjacency multilists. Once again, the choice of a particular representation will depend upon the application one has in mind and the functions one expects to perform on the graph.

```

class Graph
// objects: A nonempty set of vertices and a set of undirected edges,
// where each edge is a pair of vertices.
public:
    virtual ~Graph() {}
    // virtual destructor
    bool IsEmpty() const {return n == 0;}
    // return true iff graph has no vertices
    int NumberOfVertices() const {return n;}
    // return number of vertices in the graph
    int NumberOfEdges() const {return e;}
    // return number of edges in the graph
    virtual int Degree(int u) const = 0;
    // return number of edges incident to vertex u
    virtual bool ExistsEdge(int u, int v) const = 0;
    // return true iff graph has the edge (u,v)
    virtual void InsertVertex(int v) = 0;
    // insert vertex v into graph; v has no incident edges
    virtual void InsertEdge(int u, int v) = 0;
    // insert edge (u,v) into graph
    virtual void DeleteVertex(int v) = 0;
    // delete v and all edges incident to it
    virtual void DeleteEdge(int u, int v) = 0;
    // delete edge (u, v) from the graph
private:
    int n;    // number of vertices
    int e;    // number of edges
};

```

ADT 6.1: Abstract data type *Graph*

6.1.3.1 Adjacency Matrix

Let $G = (V, E)$ be a graph with n vertices, $n \geq 1$. The adjacency matrix of G is a two-dimensional $n \times n$ array, say a , with the property that $a[i][j] = 1$ iff the edge (i, j) ($\langle i, j \rangle$ for a directed graph) is in $E(G)$. $a[i][j] = 0$ if there is no such edge in G . The adjacency matrices for the graphs G_1 , G_2 , and G_4 are shown in Figure 6.7. The adjacency matrix for an undirected graph is symmetric, as the edge (i, j) is in $E(G)$ iff the edge (j, i) is also in $E(G)$. The adjacency matrix for a directed graph may not be symmetric (as is the case for G_1). The space needed

332 Graphs

to represent a graph using its adjacency matrix is n^2 bits. About half this space can be saved in the case of undirected graphs by storing only the upper or lower triangle of the matrix.



Figure 6.7: Adjacency matrices

From the adjacency matrix, one may readily determine if there is an edge connecting any two vertices i and j . For an undirected graph the degree of any vertex i is its row sum:

$$\sum_{j=0}^{n-1} a[i][j]$$

For a directed graph the row sum is the out-degree, and the column sum is the in-degree.

Suppose we want to answer a nontrivial question about graphs, such as, How many edges are there in G ? or, Is G connected? Adjacency matrices will require at least $O(n^2)$ time, as $n^2 = n$ entries of the matrix (diagonal entries are zero) have to be examined. When graphs are sparse (i.e., most of the terms in the adjacency matrix are zero) one would expect that the former question could be answered in significantly less time, say $O(e + n)$, where e is the number of edges in G , and $e \ll n^2/2$. Such a speed-up can be made possible through the use of a representation in which only the edges that are in G are explicitly stored. This leads to the next representation for graphs, adjacency lists.

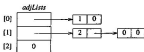
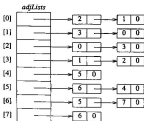
(a) G_1 (b) G_2 (c) G_4

Figure 6.8: Adjacency lists

6.1.3.2 Adjacency Lists

In this representation of graphs, the n rows of the adjacency matrix are represented as n chains (though sequential lists could be used just as well). There is one chain for each vertex in G . The nodes in chain i represent the vertices that are adjacent from vertex i . The data field of a chain node stores the index of an adjacent vertex. The adjacency lists for G_1 , G_2 , and G_4 are shown in Figure 6.8. Notice that the vertices in each chain are not required to be ordered. An array `adjLists` is used so that we can access the adjacency list for any vertex in $O(1)$ time. This array may be declared as

```
Chain<int> *adjLists;
```

where `Chain` is the template chain class of Chapter 4. If `LinkedListGraph` is our class for undirected graphs represented as linked adjacency lists as in Figure 6.8, then `adjLists` is a private data member of this class and the class constructor is as below.

```
LinkedListGraph(const int vertices = 0) : n(vertices), e(0)
{adjLists = new Chain<int>[n];}
```

For an undirected graph with n vertices and e edges, the linked adjacency lists representation requires an array of size n and $2e$ chain nodes. Each chain node has two fields. In terms of the number of bits of storage needed, the node count should be multiplied by $\log n$ for the array positions and $\log n + \log e$ for the chain nodes, as it takes $O(\log ne)$ bits to represent a number of value n . If instead of chains, we use sequential lists, the adjacency lists may be packed into an integer array `node[n + 2e + 1]`. In one possible sequential mapping, `node[i]` gives the starting point of the list for vertex i , $0 \leq i < n$, and `node[n]` is set to $n + 2e + 1$. The vertices adjacent from vertex i are stored in `node[i]`, ..., `node[i + 1] - 1`, $0 \leq i < n$. Figure 6.9 shows the representation for the graph G_4 of Figure 6.5.

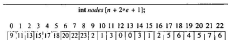


Figure 6.9: Sequential representation of graph G_4 .

The degree of any vertex in an undirected graph may be determined by just counting the number of nodes in its adjacency list.

For a digraph, the number of list nodes is only e . The out-degree of any vertex may be determined by counting the number of nodes on its adjacency list. Determining the in-degree of a vertex is a little more complex. If there is a need to access repeatedly all vertices adjacent to another vertex, then it may be worth the effort to keep another set of lists in addition to the adjacency lists. This set of lists, called *inverse adjacency lists*, will contain one list for each vertex. Each list will contain a node for each vertex adjacent to the vertex it represents (see Figure 6.10).



Figure 6.10: Inverse adjacency lists for G_3 (Figure 6.2(c))

Alternatively, one can adopt a simplified version of the list structure used for sparse matrix representation in Chapter 4. Figure 6.11 shows the resulting structure for the graph G_3 of Figure 6.2(c). The header nodes are stored sequentially. The first two fields in each node give the head and tail of the edge represented by the node, the remaining two fields are links for row and column chains.

6.1.3.3 Adjacency Multilists

In the adjacency-list representation of an undirected graph, each edge (u,v) is represented by two entries, one on the list for u and the other on the list for v . As we shall see, in some situations it is necessary to be able to determine the second entry for a particular edge and mark that edge as having been examined. This can be accomplished easily if the adjacency lists are actually maintained as multilists (i.e., lists in which nodes may be shared among several lists). For each edge there will be exactly one node, but this node will be in two lists (i.e., the adjacency lists for each of the two nodes to which it is incident). The new node structure is

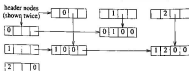


Figure 6.11: Orthogonal list representation for G_3 of Figure 6.2(c)

m	vertex1	vertex2	link1	link2
-----	---------	---------	-------	-------

where m is a Boolean mark field that may be used to indicate whether or not the edge has been examined. The storage requirements are the same as for normal adjacency lists, except for the addition of the mark bit m . Figure 6.12 shows the adjacency multilists for G_1 of Figure 6.2(a).

6.1.3.4 Weighted Edges

In many applications, the edges of a graph have weights assigned to them. These weights may represent the distance from one vertex to another or the cost of going from one vertex to an adjacent vertex. In these applications, the adjacency matrix entries $a[i][j]$ would keep this information too. When adjacency lists are used, the weight information may be kept in the list nodes by including an additional field, *weight*. A graph with weighted edges is called a *network*.

6.1.3.5 C++ Graph Classes

There are two graph types—undirected and directed. A graph of each type may be weighted or unweighted and may be represented using any of the four methods—matrix, linked adjacency lists, sequential adjacency lists, adjacency multilists. So, to accommodate all of these possibilities, we must implement 16 graph classes. For example, the class *LinkedGraph* could be for unweighted undirected graphs using linked adjacency lists and the class *MatrixWDigraph*

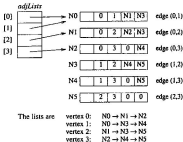


Figure 6.12: Adjacency multilists for G_1 of Figure 6.3(a)

could be for weighted directed graphs using a matrix representation. Since many of the functions (e.g., return the number of vertices and edges) we perform on graphs are independent of the graph type and representation, we can reduce the coding effort by defining a virtual class *Graph* (as in ADT 6.1) that is the repository for common codes. Classes such as *LinkedGraph* and *MatrixWDGraph* can then be derived (directly or indirectly) from *Graph*, thereby inheriting the common functions. The classes for networks (i.e., graphs with weighted edges) may be template classes or we may restrict the edge weight to be of type *double*. Figure 6.13 shows a possible derivation hierarchy for our graph classes. This figure includes only classes that use the matrix and linked adjacency list representations. Classes for other representations may be added to the hierarchy.

In the remaining sections of this chapter, we investigate several interesting problems defined on graphs. For convenience, this investigation, at times, assumes a definite representation for a graph. In many cases, the code for the corresponding function can be written in an implementation independent manner provided we have an iterator for each non-abstract class that derives from *Graph*. In these cases, the implementation-independent code may be physically placed in the abstract class *Graph* and used by all implementations. In fact, we may

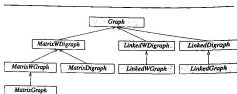


Figure 6.13: Possible derivation hierarchy for graph classes

additionally write customized codes for some of the implementations. For example, we may write a customized code for some function *foo* in the class *MatrixGraph* and also have an implementation independent code for *foo* (using iterators) in the class *Graph*. On graphs of type *MatrixGraph*, the custom code is used while on graph types for which there is no custom code, the generic code in *Graph* is used. Custom implementation for *foo*, of course, would make sense only if the custom implementation has beneficial attributes (e.g., it runs faster) relative to the implementation independent code.

EXERCISES

1. Does the multigraph of Figure 6.14 have an Eulerian walk? If so, find one.



Figure 6.14: A multigraph

2. For the digraph of Figure 6.15 obtain
 - (a) the in-degree and out-degree of each vertex
 - (b) its adjacency-matrix
 - (c) its adjacency-list representation
 - (d) its adjacency-multilist representation
 - (e) its strongly connected components



Figure 6.15: A digraph

3. Draw the complete undirected graphs on one, two, three, four, and five vertices. Prove that the number of edges in an n -vertex complete graph is $n(n-1)/2$.
4. Is the directed graph of Figures 6.16 strongly connected? List all the simple paths.



Figure 6.16: A directed graph

340 Graphs

5. Obtain the adjacency-matrix, adjacency-list, and adjacency-multilist representations of the graph of Figure 6.16.
6. Show that the sum of the degrees of the vertices of an undirected graph is twice the number of edges.
7. (a) Let G be a connected, undirected graph on n vertices. Show that G must have at least $n - 1$ edges and that all connected, undirected graphs with $n - 1$ edges are trees.
(b) What is the minimum number of edges in a strongly connected digraph on n vertices? What form do such digraphs have?
8. For an undirected graph G with n vertices, prove that the following are equivalent:
 - (a) G is a tree
 - (b) G is connected, but if any edge is removed the resulting graph is not connected
 - (c) For any two distinct vertices $u \in V(G)$ and $v \in V(G)$, there is exactly one simple path from u to v
 - (d) G contains no cycles and has $n - 1$ edges
9. Write a C++ function to input the number of vertices and edges in an undirected graph. Next, input the edges one by one and to set up the linked adjacency-list representation of the graph. You may assume that no edge is input twice. What is the run time of your function as a function of the number of vertices and the number of edges?
10. Do the preceding exercise but this time set up the multilist representation.
11. Let G be an undirected, connected graph with at least one vertex of odd degree. Show that G contains no Eulerian walk.
12. [Programming Project] Write C++ code for each of the classes in the hierarchy of Figure 6.13. Each class should derive publicly from its parent class in the hierarchy. Your code for `Graph` should build upon that of ADT 6.1. For weighted graphs, assume the edge weight is of type `double`.

6.2 ELEMENTARY GRAPH OPERATIONS

When we discussed binary trees in Chapter 5, we indicated that tree traversals were among the most frequently used tree operations. Thus, we defined and implemented preorder, inorder, postorder, and level-order tree traversals. An analogous situation occurs in the case of graphs. Given a graph $G = (V, E)$ and a vertex v in $V(G)$, we wish to visit all vertices in G that are reachable from v (i.e., all vertices that are connected to v). We shall look at two ways of doing this:

depth-first search and *breadth-first search*. Although these methods work on both directed and undirected graphs, the following discussion assumes that the graphs are undirected.

6.2.1 Depth-First Search

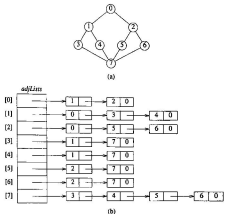
We begin by visiting the start vertex v . Next an unvisited vertex w adjacent to v is selected, and a depth-first search from w is initiated. When a vertex u is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited that has an unvisited vertex w adjacent to it and initiate a depth-first search from w . The search terminates when no unvisited vertex can be reached from any of the visited vertices. This function is best described recursively as in Program 6.1.

```
virtual void Graph::DFS() // Driver
{
    visited = new bool [n];
    // visited is declared as a bool* data member of Graph
    fill(visited, visited + n, false);
    DFS(0); // start search at vertex 0
    delete [] visited;
}

virtual void Graph::DFS(const int v) // Workhorse
{// Visit all previously unvisited vertices that are reachable from vertex v.
    visited[v] = true;
    for (each vertex w adjacent to v) // actual code uses an iterator
        if (!visited[w]) DFS(w);
}
```

Program 6.1: Depth-first search

Example 6.1: Consider the graph G of Figure 6.17(a), which is represented by its adjacency lists as in Figure 6.17(b). If a depth-first search is initiated from vertex 0, then the vertices of G are visited in the following order: 0, 1, 3, 7, 4, 5, 2, 6. Since $DFS(0)$ visits all vertices that can be reached from 0, the vertices visited, together with all edges in G incident to these vertices, form a connected component of G . \square

Figure 6.17: Graph G and its adjacency list

Analysis of DFS: When G is represented by its adjacency lists, the vertices w adjacent to v can be determined by following a chain of links. Since *DFS* examines each node in the adjacency lists at most once, and there are $2e$ list nodes, the time to complete the search is $O(e)$. If G is represented by its adjacency matrix, then the time to determine all vertices adjacent to v is $O(n)$. Since at most n vertices are visited, the total time is $O(n^2)$. \square

6.2.2 Breadth-First Search

In a breadth-first search, we begin by visiting the start vertex v . Next, all unvisited vertices adjacent to v are visited. Unvisited vertices adjacent to these newly visited vertices are then visited, and so on. Algorithm *BFS* (Program 6.2) gives the details.

Example 6.2: Consider the graph of Figure 6.17(a). If a breadth-first search is performed beginning at vertex 0, then we first visit vertex 0. Next, vertices 1 and 2 are visited. Then vertices 3, 4, 5, and 6, and finally 7, are visited. \square

```

virtual void Graph::BFS(int v)
// A breadth first search of the graph is carried out beginning at vertex v.
// visited[i] is set to true when i is visited. The function uses a queue.
    visited = new bool [n];
    fill(visited, visited + n, false);
    visited[v] = true;
    Queue<int> q;
    q.Push(v);
    while (!q.IsEmpty()) {
        v = q.Front();
        q.Pop();
        for (all vertices w adjacent to v) // actual code uses an iterator
            if (!visited[w]) {
                q.Push(w);
                visited[w] = true;
            }
    } // end of while loop
    delete [] visited;
}

```

Program 6.2: Breadth-first search

Analysis of *BFS*: Each visited vertex enters the queue exactly once. So, the while loop is iterated at most n times. If an adjacency matrix is used, the loop takes $O(n)$ time for each vertex visited. The total time is, therefore, $O(n^2)$. If adjacency lists are used, the loop has a total cost of $d_0 + \cdots + d_{n-1} = O(e)$, where d_i is the degree of vertex i . As in the case of *DFS*, all visited vertices, together with all edges incident to them, form a connected component of G . \square

6.2.3 Connected Components

If G is an undirected graph, then one can determine whether or not it is connected by simply making a call to either *DFS* or *BFS* and then determining if there is any unvisited vertex. The connected components of a graph may be obtained by making repeated calls to either *DFS*(v) or *BFS*(v), where v is a vertex that has not yet been visited. This leads to function *Components* (Program 6.3), which determines the connected components of G . The algorithm uses *DFS* (*BFS* may be used instead if desired). The computing time is not affected. Function *Graph::OutputNewComponent* outputs all vertices visited in the most recent invocation of *DFS*, together with all edges incident on these vertices.

```
virtual void Graph::Components()
// Determine the connected components of the graph.
// visited is assumed to be declared as a bool* data member of Graph
visited = new bool [n];
for (visited, visited + n, false);
for (i = 0; i < n; i++)
    if (!visited[i]) {
        DFS(i); // find a component
        OutputNewComponent();
    }
delete [] visited;
}
```

Program 6.3: Determining connected components

Analysis of Components: If G is represented by its adjacency lists, then the total time taken by *DFS* is $O(e)$. The output can be completed in time $O(e)$ if *DFS* keeps a list of all newly visited vertices. Since the for loops take $O(n)$ time, the total time to generate all the connected components is $O(n + e)$. If adjacency matrices are used instead, the time required is $O(n^2)$. \square

6.2.4 Spanning Trees

When the graph G is connected, a depth-first or breadth-first search starting at any vertex visits all the vertices in G . In this case the edges of G are partitioned into two sets, T (for tree edges) and N (for nontree edges), where T is the set of

edges used or traversed during the search and N the set of remaining edges. The set T may be determined by inserting the statement $T = T \cup \{(v, w)\}$ in the if clauses of *DFS* and *BFS*. The edges in T form a tree that includes all the vertices of G . Any tree consisting solely of edges in G and including all vertices in G is called a *spanning tree*. Figure 6.18 shows a graph and some of its spanning trees.

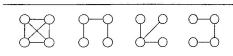


Figure 6.18: A complete graph and three of its spanning trees

As indicated earlier, a spanning tree may be constructed using either a depth-first or a breadth-first search. The spanning tree resulting from a depth-first search is known as a *depth-first spanning tree*. When a breadth-first search is used, the spanning tree is called a *breadth-first spanning tree*. Figure 6.19 shows the spanning trees resulting from a depth-first and breadth-first search starting at vertex 0 of the graph of Figure 6.17.



Figure 6.19: Depth-first and breadth-first spanning trees for graph of Figure 6.17

If a nontree edge (v, w) is introduced into any spanning tree T , then a cycle

is formed. This cycle consists of the edge (v, w) and all the edges on the path from w to v in T . For example, if the edge $(7, 6)$ is introduced into the DFS spanning tree of Figure 6.19(a), then the resulting cycle is 7, 6, 2, 5, 7. We can use this property of spanning trees to obtain an independent set of circuit equations for an electrical network.

Example 6.3 [Creation of circuit equations]: To obtain the circuit equations, we must first obtain a spanning tree for the electrical network. Then we introduce the nontree edges into the spanning tree one at a time. The introduction of each such edge produces a cycle. Next we use Kirchhoff's second law on this cycle to obtain a circuit equation. The cycles obtained in this way are independent (we cannot obtain any of these cycles by taking a linear combination of the remaining cycles), since each contains a nontree edge that is not contained in any other cycle. Thus, the circuit equations are also independent. In fact, we can show that the cycles obtained by introducing the nontree edges one at a time into the spanning tree form a cycle basis. This means that we can construct all other cycles in the graph by taking a linear combination of the cycles in the basis. \square

Let us examine a second property of spanning trees. A spanning tree is a *minimal subgraph*, G^* , of G such that $V(G^*) = V(G)$, and G^* is connected. We define a *minimal subgraph* as one with the fewest number of edges. Any connected graph with n vertices must have at least $n - 1$ edges, and all connected graphs with $n - 1$ edges are trees. Therefore, we conclude that a spanning tree has $n - 1$ edges. (The Exercises explore this property more fully.)

Constructing minimal subgraphs finds frequent application in the design of communication networks. Suppose that the vertices of a graph G represent cities, and the edges represent communication links between cities. The minimum number of links needed to connect n cities is $n - 1$. Constructing the spanning trees of G produces all feasible choices. However, we know that the cost of constructing communication links between cities is rarely the same. Therefore, in practical applications, we assign weights to the edges. These weights might represent the cost of constructing the communication link or the length of the link. Given such a weighted graph, we would like to select the spanning tree that represents either the lowest total cost or the lowest overall length. We assume that the cost of a spanning tree is the sum of the costs of the edges of that tree. Algorithms to obtain minimum-cost spanning trees are studied in Section 6.3.

6.2.5 Biconnected Components

The operations that we have implemented thus far are simple extensions of depth-first and breadth-first searches. The next operation we implement is more complex and requires the introduction of additional terminology. We begin by assuming that G is an undirected, connected graph.

Definition: A vertex v of G is an *articulation point* iff the deletion of v , together with the deletion of all edges incident to v , leaves behind a graph that has at least two connected components. \square

Vertices 1, 3, 5, and 7 are the articulation points of the connected graph of Figure 6.20(a).

Definition: A *biconnected graph* is a connected graph that has no articulation points. \square

The graph of Figure 6.20 is not biconnected, and that of Figure 6.17(a) is. Articulation points are undesirable in graphs that represent communication networks. In such graphs the vertices represent communication stations, and the edges represent communication links. The failure of a communication station that is an articulation point results in a loss of communication to stations other than the one that failed. If the communication graph is biconnected, then the failure of a single station results in a loss of communication to and from only that station.

Definition: A *biconnected component* of a connected graph G is a maximal biconnected subgraph H of G . By maximal, we mean that G contains no other subgraph that is both biconnected and properly contains H . \square

The graph of Figure 6.20(a) contains six biconnected components. These are shown in Figure 6.20(b). Note that a biconnected graph has just one biconnected component: the whole graph. It is easy to verify that two biconnected components of the same graph can have at most one vertex in common. From this it follows that no edge can be in two or more biconnected components. Hence, the biconnected components of G partition the edges of G .

The biconnected components of a connected, undirected graph G can be found by using any depth-first spanning tree of G . For the graph of Figure 6.20(a) a depth-first spanning tree with root 3 is shown in Figure 6.21(a). This tree is redrawn in Figure 6.21(b) to better reveal the tree structure. This figure also shows the nontree edges of G by broken lines. The numbers outside the vertices give the sequence in which the vertices are visited during the depth-first

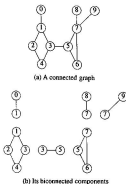


Figure 6.26: A connected graph and its biconnected components

search. This number is called the *depth-first number*, dfn , of the vertex. So, $dfn(0) = 3$, and $dfn(9) = 9$. Note that if u and v are two vertices such that u is an ancestor of v in the depth-first spanning tree, then $dfn(u) < dfn(v)$.

The broken lines in Figure 6.21(b) represent nontree edges. A nontree edge (u, v) is a *back edge* with respect to a spanning tree T iff either u is an ancestor of v or v is an ancestor of u . A nontree edge that is not a back edge is called a *cross edge*. The nontree edges $(3, 1)$ and $(5, 7)$ are back edges. From the definition of a depth-first search, one can show that no graph can have cross edges with respect to any of its depth-first spanning trees. From this, it follows that the root of the depth-first spanning tree is an articulation point iff it has at least two children. Further, any other vertex u is an articulation point iff it has at least one child, w , such that it is not possible to reach an ancestor of w using a path

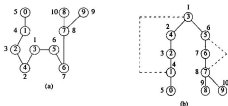


Figure 6.21: Depth-first spanning tree of Figure 6.20(a)

composed solely of w , descendants of w , and a single back edge. These observations lead us to define a value low for each vertex of G such that $low(w)$ is the lowest depth-first number that can be reached from w using a path of descendants followed by, at most, one back edge. $low(w)$ is given by the equation

$$low(w) = \min\{dfn(w), \min\{low(x) \mid x \text{ is a child of } w\}, \min\{dfn(x) \mid (w, x) \text{ is a back edge}\}\}$$

From the preceding discussion it follows that u is an articulation point iff u is either the root of the spanning tree and has two or more children or u is not the root and u has a child w such that $low(w) \geq dfn(u)$. Figure 6.22 gives the dfn and low values for each vertex of the spanning tree of Figure 6.21(b).

vertex	0	1	2	3	4	5	6	7	8	9
dfn	5	4	3	1	2	6	7	8	10	9
low	5	1	1	1	1	6	6	6	10	9

Figure 6.22: dfn and low values for the spanning tree of Figure 6.21(b)

Function *DFS* is easily modified to compute dfn and low for each vertex of

350 Graphs

a connected graph. The result is function *DfnLow* (Program 6.4). This function uses the function *min*, which returns the smaller of its two parameters.

```
virtual void Graph::DfnLow(const int x) // begin DFS at vertex x
{
    num = 1;                // num is an int data member of Graph
    dfn = new int[n];        // dfn is declared as int* in Graph
    low = new int[n];        // low is declared as int* in Graph
    fill(dfn, dfn + n, 0);
    fill(low, low + n, 0);
    DfnLow(x, -1); // start at vertex x
    delete [] dfn;
    delete [] low;
}

void Graph::DfnLow (const int u, const int v)
// Compute dfn and low while performing a depth first search beginning at vertex
// u. v is the parent (if any) of u in the resulting spanning tree.
    dfn[u] = low[u] = num++;
    for (each vertex w adjacent from u) // actual code uses an iterator
        if (dfn[w] == 0) { // w is an unvisited vertex
            DfnLow(w, u);
            low[u] = min(low[u], low[w]);
        }
    else if (v == w) low[u] = min(low[u], dfn[w]); // back edge
}
```

Program 6.4: Computing *dfn* and *low*

The edges of the connected graph may be partitioned into their biconnected components by adding some code to function *DfnLow*. First, note that following the return from *DfnLow*(*w*, *u*), *low*[*w*] has been computed. If *low*[*w*] \geq *dfn*[*u*], then a new biconnected component has been identified. By using a stack to save edges when they are first encountered, we can output all edges in a biconnected component, as in function *Biconnected* (Program 6.5).

Establishing the correctness of function *Biconnected* is left as an exercise. Its complexity is $O(n + e)$. Note that function *Biconnected* assumes that the input connected graph has at least two vertices. Connected graphs with just one vertex contain no edges. By convention these graphs are biconnected, and a proper biconnected components function should handle them as a special case, producing a single biconnected component as output.

```

virtual void Graph::Biconnected ()
{
    num = 1;           // num is an int data member of Graph
    dfn = new int[n];   // dfn is declared as int* in Graph
    low = new int[n];   // low is declared as int* in Graph
    fill(dfn, dfn + n, 0);
    fill(low, low + n, 0);
    Biconnected(0, -1); // start at vertex 0
    delete [] dfn;
    delete [] low;
}

virtual void Graph::Biconnected (const int u, const int v)
{
    // Compute dfn and low, and output the edges of G by their biconnected
    // components. v is the parent (if any) of u in the resulting spanning tree.
    // s is an initially empty stack declared as a data member of Graph.
    dfn[u] = low[u] = num++;
    for ( each vertex w adjacent from u ) { // actual code uses an iterator
        if ((v != w) && (dfn[w] < dfn[u])) add (u, w) to stack s;
        if (dfn[w] == 0) { // w is an unvisited vertex
            Biconnected(w, u);
            low[u] = min(low[u], low[w]);
            if (low[w] >= dfn[u]) {
                cout << "New Biconnected Component: " << endl;
                do {
                    delete an edge from the stack s;
                    let this edge be (x, y);
                    cout << x << ", " << y << endl;
                } while ( (x, y) and (u, w) are not the same edge )
            }
        }
        else if (w != v) low[u] = min(low[u], dfn[w]); // back edge
    }
}

```

Program 6.5: Outputting biconnected components when $n > 1$

EXERCISES

1. Apply depth-first and breadth-first searches to the complete graph on four vertices. List the vertices in the order they would be visited.
2. Write a complete C++ function for depth-first search under the assumption that graphs are represented using adjacency matrices. Test the correctness of your function using suitable graphs.
3. Write a complete C++ function for depth-first search under the assumption that graphs are represented using adjacency lists. Test the correctness of your function using suitable graphs.
4. Write a complete C++ function for breadth-first search under the assumption that graphs are represented using adjacency matrices. Test the correctness of your function using suitable graphs.
5. Write a complete C++ function for breadth-first search under the assumption that graphs are represented using adjacency lists. Test the correctness of your function using suitable graphs.
6. Show how to modify function *DFS* (Program 6.1), as it is used in *Components* (Program 6.3), to produce a list of all newly visited vertices.
7. Prove that when function *DFS* (Program 6.1) is applied to a connected graph, the edges of T form a tree.
8. Prove that when function *BFS* (Program 6.2) is applied to a connected graph, the edges of T form a tree.
9. Show that if T is a spanning tree for the undirected graph G , then the addition of an edge e , $e \notin E(T)$ and $e \in E(G)$, to T creates a unique cycle.
10. Show that the number of spanning trees in a complete graph with n vertices is at least $2^{n-1} - 1$.
11. Let G be a connected graph and let T be any of its depth-first spanning trees. Show that G has no cross edges relative to T .
12. Prove that function *Biconnected* (Program 6.5) correctly partitions the edges of a connected graph into the biconnected components of the graph.
13. Let G be a connected, undirected graph. Show that no edge of G can be in two or more biconnected components of G .

6.3 MINIMUM-COST SPANNING TREES

The *cost* of a spanning tree of a weighted, undirected graph is the sum of the costs (weights) of the edges in the spanning tree. A *minimum-cost spanning tree* is a spanning tree of least cost. Three different algorithms can be used to obtain

a minimum-cost spanning tree of a connected, undirected graph. All three use a design strategy called the *greedy method*. We shall refer to the three algorithms as Kruskal's, Prim's, and Sollin's algorithms, respectively.

In the greedy method, we construct an optimal solution in stages. At each stage, we make the best decision (using some criterion) possible at the time. Since we cannot change this decision later, we make sure that the decision will result in a feasible solution. The greedy method can be applied to a wide variety of programming problems. Typically, the selection of an item at each stage is based on either a least-cost or a highest profit criterion. A feasible solution is one that works within the constraints specified by the problem.

To construct minimum-cost spanning trees, we use a least-cost criterion. Our solution must satisfy the following constraints:

- (1) We must use only edges within the graph.
- (2) We must use exactly $n - 1$ edges.
- (3) We may not use edges that produce a cycle.

6.3.1 Kruskal's Algorithm

Kruskal's algorithm builds a minimum-cost spanning tree T by adding edges to T one at a time. The algorithm selects the edges for inclusion in T in nondecreasing order of their cost. An edge is added to T if it does not form a cycle with the edges that are already in T . Since G is connected and has $n > 0$ vertices, exactly $n - 1$ edges will be selected for inclusion in T .

Example 6.4: We will construct a minimum-cost spanning tree of the graph of Figure 6.23(a). We begin with no edges selected. Figure 6.23(b) shows the current graph with no edges selected. Edge (0,5) is the first edge considered. It is included in the spanning tree being built. This yields the graph of Figure 6.23(c).

Next, the edge (2,3) is selected and included in the tree (Figure 6.23(d)). The next edge to be considered is (1,6). Its inclusion in the tree being built does not create a cycle, so we get the graph of Figure 6.23(e). Edge (1,2) is considered next and included in the tree (Figure 6.23(f)). Of the edges not yet considered, (6,3) has the least cost. It is considered next. Its inclusion in the tree results in a cycle, so this edge is discarded. Edge (4,3) is the next edge to be added to the tree being built. This results in the configuration of Figure 6.23(g). The next edge to be considered is the edge (6,4). It is discarded, as its inclusion creates a cycle. Finally, edge (5,4) is considered and included in the tree being built. This completes the spanning tree. The resulting tree (Figure 6.23(h)) has cost 99. □

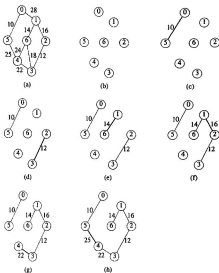


Figure 6.23: Stages in Kruskal's algorithm

It is somewhat surprising that this straightforward approach should always result in a minimum-cost spanning tree. We shall soon prove, however, that this is indeed the case. First, let us look into the details of the algorithm. For clarity, Kruskal's algorithm is written out more formally in Program 6.6. Initially, E is the set of all edges in G . The only functions we wish to perform on this set are

- (1) determine an edge with minimum cost (line 3)
- (2) delete this edge (line 4)

Both these functions can be performed efficiently if the edges in E are maintained as a sorted sequential list. In Chapter 7 we shall see how to sort these edges into nondecreasing order in time $O(e \log e)$, where e is the number of edges in E . It is not essential to sort all the edges, as long as the next edge for line 3 can be determined easily. This is an instance where a min heap is ideal, as it permits the next edge to be determined and deleted in $O(\log e)$ time. The construction of the heap itself takes $O(e)$ time.

```

1  $T \leftarrow \emptyset$ ;
2 while (( $T$  contains less than  $n - 1$  edges) && ( $E$  not empty)) {
3   choose an edge  $(v, w)$  from  $E$  of lowest cost;
4   delete  $(v, w)$  from  $E$ ;
5   if ( $(v, w)$  does not create a cycle in  $T$ ) add  $(v, w)$  to  $T$ ;
6   else discard  $(v, w)$ ;
7 }
8 if ( $T$  contains fewer than  $n - 1$  edges) cout << "no spanning tree" << endl;

```

Program 6.6: Kruskal's algorithm

To perform line 5 of Program 6.6 efficiently, the vertices in G should be grouped together in such a way that one may easily determine if the vertices v and w are already connected by the earlier selection of edges. If they are, then the edge (v, w) is to be discarded. If they are not, then (v, w) is to be added to T . One possible grouping is to place all vertices in the same connected component of T into a set (all connected components of T will also be trees). Then, two vertices v and w are connected in T iff they are in the same set. For example, when the edge $(3, 6)$ is to be considered, the sets would be $\{0, 5\}$, $\{1, 2, 3, 6\}$, and $\{4\}$. Vertices 3 and 6 are already in the same set, so the edge $(3, 6)$ is rejected. The next edge to be considered is $(3, 4)$. Since vertices 3 and 4 are in different sets, the edge is accepted. This edge connects the two components $\{1, 2, 3, 6\}$ and $\{4\}$, so these two sets should be unioned to obtain the set representing the new component. Using the set representation scheme of Chapter 5, we can obtain an

efficient implementation of line 5. The computing time is, therefore, determined by the time for lines 3 and 4, which in the worst case is $O(e \log e)$. We leave the writing of the resulting algorithm as an exercise. Theorem 6.1 proves that the algorithm resulting from Program 6.6 does yield a minimum-cost spanning tree of G .

Theorem 6.1: Let G be any undirected, connected graph. Kruskal's algorithm generates a minimum-cost spanning tree.

Proof: We shall show the following: (a) Kruskal's method results in a spanning tree whenever a spanning tree exists; and (b) the spanning tree generated is of minimum cost.

For (a), we note that the only edges that get discarded in Kruskal's method are those that result in a cycle. The deletion of a single edge that is on a cycle of a connected graph results in a graph that is also connected. Hence, if G initially is connected, the set of edges in T and E always form a connected graph. Consequently, if G initially is connected, the algorithm cannot terminate with $E = \emptyset$ and $|T| < n - 1$.

Now, let us proceed to establish that the constructed spanning tree, T , is of minimum cost. Since G has a finite number of spanning trees, it must have at least one of minimum cost. Let U be a minimum-cost spanning tree. Both T and U have exactly $n-1$ edges. If $T = U$, then T is of minimum cost, and we have nothing to prove. So, assume that $T \neq U$. Let $k, k > 0$, be the number of edges in T that are not in U . Note that k is also the number of edges in U that are not in T .

We shall show that T and U have the same cost by transforming U into T . This transformation will be done in k steps. At each step, the number of edges in T that are not in U will be reduced by exactly 1. Further, the cost of U will not change as a result of the transformation. As a result, U after k steps of transformation will have the same cost as the initial U and will consist of exactly those edges that are in T . This implies that T is of minimum cost.

Each step of the transformation involves adding to U one edge, e , from T and removing one edge, f , from U . The edges e and f are selected in the following way:

- (1) Let e be the least-cost edge in T that is not in U . Such an edge must exist as $k > 0$.
- (2) When e is added to U , a unique cycle is created. Let f be any edge on this cycle that is not in T . Note that at least one of the edges on this cycle is not in T , as T contains no cycles.

From the way e and f are selected, it follows that $V = U + \{e\} - \{f\}$ is a spanning tree and that T has exactly $k-1$ edges that are not in V . We need to

show that the cost of V is the same as that of U . Clearly, the cost of V is the cost of U plus the cost of the edge e minus the cost of the edge f . The cost of e cannot be less than the cost of f , as otherwise the spanning tree V has a smaller cost than the tree U , which is impossible. If e has a higher cost than f , then f is considered before e by Kruskal's algorithm. Since f is not in T , Kruskal's algorithm must have discarded this edge at this time. Hence, f , together with edges in T having a cost less than or equal to the cost of f , must form a cycle. By the choice of e , all these edges are also in U . Hence, U must also contain a cycle. But it does not, as it is a spanning tree. So, the assumption that e is of higher cost than f leads to a contradiction. The only possibility that remains is that e and f have the same cost. Hence, V has the same cost as U . \square

6.3.2 Prim's Algorithm

Prim's algorithm, like Kruskal's, constructs the minimum-cost spanning tree edge by edge. However, at all times during the algorithm, the set of selected edges forms a tree. (By contrast, the set of selected edges in Kruskal's algorithm forms a forest at all times.) Prim's algorithm begins with a tree T that contains a single vertex. This vertex can be any of the vertices in the original graph. Then we add a least-cost edge (u, v) to T such that $T \cup \{(u, v)\}$ is also a tree. This edge-addition step is repeated until T contains $n-1$ edges. Notice that edge (u, v) is always such that exactly one of u and v is in T . A high-level description of Prim's algorithm is provided in Program 6.7. This description also provides for the possibility that the input graph may not be connected. In this case there is no spanning tree. Figure 6.24 shows the progress of Prim's algorithm on the graph of Figure 6.23(a).

```
// Assume that  $G$  has at least one vertex.
TV = {0}; // start with vertex 0 and no edges
for ( $T = \emptyset$ ;  $T$  contains fewer than  $n-1$  edges; add  $(u, v)$  to  $T$ )
{
    Let  $(u, v)$  be a least-cost edge such that  $u \in TV$  and  $v \notin TV$ ;
    if (there is no such edge) break;
    add  $v$  to  $TV$ ;
}
if ( $T$  contains fewer than  $n-1$  edges) cout << "no spanning tree" << endl;
```

Program 6.7: Prim's algorithm

Prim's algorithm can be implemented to have a time complexity $O(n^2)$ if

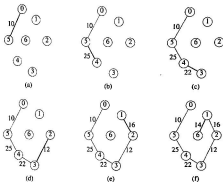


Figure 6.24: Stages in Prim's algorithm

we associate with each vertex v not in TV a vertex $\text{near}(v)$ such that $\text{near}(v) \in TV$ and $\text{cost}(\text{near}(v), v)$ is the minimum over all such choices for $\text{near}(v)$ (we assume that $\text{cost}(v, w) = \infty$ if $(v, w) \notin E$). The next edge to add to T is such that $\text{cost}(\text{near}(v), v)$ is the minimum and $v \notin TV$. Asymptotically faster implementations are also possible. One of these results from the use of Fibonacci heaps, which are studied in Chapter 9. Establishing the correctness of Prim's algorithm is left as an exercise.

6.3.3 Sollin's Algorithm

Sollin's algorithm selects several edges at each stage. At the start of a stage, the selected edges, together with all n graph vertices, form a spanning forest. During a stage we select one edge for each tree in this forest. This edge is a minimum-cost edge that has exactly one vertex in the tree. The selected edges are added to the spanning tree being constructed. Note that it is possible for two trees in the forest to select the same edge. So, multiple copies of the same edge are to be eliminated. Also, when the graph has several edges with the same cost, it is possible for two trees to select two different edges that connect them together. These edges will, of course, have the same cost; only one of these should be retained. At the start of the first stage, the set of selected edges is empty. The algorithm terminates when there is only one tree at the end of a stage or when no edges remain to be selected.

Figure 6.25 shows the stages in Sollin's algorithm when it begins with the graph of Figure 6.23(a). The initial configuration of zero selected edges is the same as that shown in Figure 6.23(b). Each tree in this spanning forest is a single vertex. The edges selected by vertices 0, 1, ..., 6 are, respectively, (0, 5), (1, 6), (2, 3), (3, 2), (4, 3), (5, 0), and (6, 1). The distinct edges in this selection are (0, 5), (1, 6), (2, 3), and (4, 3). Adding these to the set of selected edges results in the configuration of Figure 6.25(a). In the next stage, the tree with vertex set {0, 5} selects the edge (5, 4), and the remaining two trees select the edge (1, 2). Following the addition of these two edges to the set of selected edges, construction of the spanning tree is complete. The resulting spanning tree is shown in Figure 6.25(b). The development of Sollin's algorithm into a C++ function and its correctness proof are left as exercises.

EXERCISES

1. Write out Kruskal's minimum-cost spanning tree algorithm (Program 6.6) as a complete program. You may use as functions the algorithms *WeightedUnion* (Program 5.22) and *CollapsingFind* (Program 5.23). Use a min-heap (Chapter 5) to select the edges in nondecreasing order by weight.
2. Prove that Prim's algorithm finds a minimum-cost spanning tree for every connected, undirected graph.
3. Refine Program 6.7 into a C++ function to find a minimum-cost spanning tree. The complexity of your function should be $O(n^3)$, where n is the number of vertices in the input graph. Show that this is the case.
4. Prove that Sollin's algorithm finds a minimum-cost spanning tree for every connected, undirected graph.

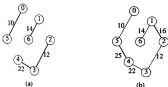


Figure 6.25: Stages in Sollin's algorithm.

5. What is the maximum number of stages in Sollin's algorithm? Give this as a function of the number of vertices n in the graph.
6. Obtain a C++ function to find a minimum-cost spanning tree using Sollin's algorithm. What is the complexity of your function?

6.4 SHORTEST PATHS AND TRANSITIVE CLOSURE

Graphs may be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway. The edges may then be assigned weights, which might be the distance between the two cities connected by the edge or the average time to drive along that section of highway. A motorist wishing to drive from city A to city B would be interested in answers to the following questions:

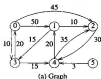
- (1) Is there a path from A to B ?
- (2) If there is more than one path from A to B , which is the shortest path?

The problems defined by (1) and (2) above are special cases of the path problems we shall be studying in this section. An edge weight is also referred to as an edge length or edge cost. We shall use the terms weight, cost, and length interchangeably. The length (cost, weight) of a path is now defined to be the sum of the lengths (costs, weights) of the edges on that path, rather than the number of edges. The starting vertex of the path will be referred to as the source and the

last vertex the *destination*. The graphs will be digraphs to allow for one-way streets.

6.4.1 Single Source/All Destinations: Nonnegative Edge Costs

In this problem we are given a directed graph $G = (V, E)$, a length function $\text{length}(i, j)$, $\text{length}(i, j) \geq 0$, for the edges of G , and a source vertex v . The problem is to determine a shortest path from v to each of the remaining vertices of G . As an example, consider the directed graph of Figure 6.26(a). The numbers on the edges are the edge lengths. If vertex 0 is the source vertex, then the shortest path from 0 to 1 is 0, 3, 4, 1. The length of this path is $10 + 15 + 20 = 45$. Even though there are three edges on this path, it is shorter than the path 0, 1, which is of length 50. There is no path from 0 to 5. Figure 6.26(b) lists the shortest paths from 0 to 1, 2, 3, and 4. The paths have been listed in nondecreasing order of path length. A greedy algorithm will generate the shortest paths in this order.



Path	Length
1) 0, 3	10
2) 0, 3, 4	25
3) 0, 3, 4, 1	45
4) 0, 2	45

(b) Shortest paths from 0

Figure 6.26: Graph and shortest paths from vertex 0 to all destinations

Let S denote the set of vertices (including the source v) to which the shortest paths have already been found. For w not in S , let $\text{dist}(w)$ be the length of the shortest path starting from v , going through only the vertices that are in S , and ending at w . We observe that when paths are generated in nondecreasing order of length,

(1) If the next shortest path is to vertex w , then the path begins at v , ends at w , and goes through only vertices that are in S . To prove this we must show that all of the intermediate vertices on the shortest path to w must be in S . Assume there is a vertex x on this path that is not in S . Then, the v -to- w path also contains a path from v to x that is of length less than that of the v -to- w path. By

assumption, the shortest paths are being generated in nondecreasing order of path length, so the shorter path from v to w has been generated already. Hence, there is no intermediate vertex that is not in S .

(2) The destination of the next path generated must be the vertex u that has the minimum distance, $\text{dist}[u]$, among all vertices not in S . This follows from the definition of dist and observation (1). If there are several vertices not in S with the same dist , then any of these may be selected.

(3) The vertex u selected in (2) becomes a member of S . The shortest v -to- u path is obtained from the selection process of (2). At this point, the length of the shortest paths starting at v , going through vertices only in S , and ending at a vertex w not in S may decrease (i.e., the value of $\text{dist}[w]$ may change). If it does change, then the change must be due to a shorter path starting at v going to u and then to w . The intermediate vertices on the v -to- u path and the u to w path must all be in S . Further, the v -to- u path must be the shortest such path; otherwise $\text{dist}[u]$ is not defined properly. Also, the u -to- w path can be chosen so that it does not contain any intermediate vertices. Therefore, we may conclude that if $\text{dist}[w]$ changes (i.e., decreases), then the change is due to the path from v to u to w , where the path from v to u is the shortest such path and the path from u to w is the edge $\langle u, w \rangle$. The length of this path is $\text{dist}[u] + \text{length}(\langle u, w \rangle)$.

The function *ShortestPath* (Program 6.8) uses these observations to determine the length of the shortest path from v to each of the other vertices in G . The generation of the paths is a minor extension of the algorithm and is left as an exercise. It is assumed that the n vertices of G are numbered 0 through $n-1$. The set S is maintained as a Boolean array with $s[i] = \text{false}$ if vertex i is not in S and $s[i] = \text{true}$ if it is. It is assumed that the graph itself is represented by its length-adjacency matrix, with $\text{length}[i][j]$ being the length of the edge $\langle i, j \rangle$. If $\langle i, j \rangle$ is not an edge of the graph and $i \neq j$, $\text{length}[i][j]$ may be set to some large number *LARGE*. The choice of this number is arbitrary, although we make two stipulations regarding its value:

- (1) The number must be larger than any of the values in the length matrix.
- (2) The number must be chosen so that the statement $\text{dist}[u] + \text{length}[u][w]$ does not produce an overflow.

Restriction (2) makes *MAXDOUBLE* (in case edge weights are of type *double*) a poor choice for nonexistent edges. For $i=j$, $\text{length}[i][j]$ may be set to any non-negative number without affecting the outcome of the function. The arrays *length*, *dist* and *s* as well as the function *Choose* used by Program 6.8 are assumed to be defined elsewhere in the class *MatrixWDDigraph* and accessible from Program 6.8.

```

1 void Main(WDigraph::ShortestPath(const int n, const int v)
2 {W.dist[j], 0 ≤ j < n, is set to the length of the shortest path from v to j
3 S is a digraph G with n vertices and edge lengths given by length[i][j].
4   for (int i = 0; i < n; i++) {s[i] = false; dist[i] = length[v][i]; S initialize
5   s[v] = true;
6   dist[v] = 0;

7   for (i = 0; i < n-2; i++) { S determine n-1 paths from vertex v
8     int u = Choose(n); S choose returns a value u such that:
9                       S dist[u] = minimum dist[w], where s[w] = false
10    s[u] = true;
11    for (int w = 0; w < n; w++)
12      if (!s[w] && dist[u] + length[u][w] < dist[w])
13        dist[w] = dist[u] + length[u][w];
14    } S end of for (i = 0; ...)
15 }
```

Program 6.8: Determining the lengths of the shortest paths

Analysis of ShortestPath: From our earlier discussion, it is easy to see that the algorithm works. The time taken by the algorithm on a graph with n vertices is $O(n^3)$. To see this, note that the for loop of line 4 takes $O(n)$ time. The for loop of line 7 is executed $n - 2$ times. Each execution of this loop requires $O(n)$ time at line 8 to select the next vertex and again in lines 11 to 13 to update *dist*. So, the total time for this loop is $O(n^2)$. If a list *T* of vertices currently not in *S* is maintained, then the number of nodes on this list would at any time be $n - i$. This would speed up lines 8 and 11 to 13, but the asymptotic time would remain $O(n^3)$. This and other variations of the algorithm are explored in the exercises.

Any shortest-path algorithm must examine each edge in the graph at least once, since any of the edges could be in a shortest path. Hence, the minimum possible time for such an algorithm would be $O(e)$. Since length-adjacency matrices were used to represent the graph, it takes $O(n^2)$ time just to determine which edges are in *G*, so any shortest-path algorithm, that uses this representation must take $O(n^3)$. For this representation, then, algorithm *ShortestPath* is optimal to within a constant factor. Even if a change to adjacency lists is made, only the overall time for the for loop of lines 11 to 13 can be brought down to $O(e)$ (since the *dist* can change only for vertices that are adjacent from *u*). The total time for line 8 remains $O(n^2)$. By using Fibonacci heaps (see Chapter 9) and adjacency lists, the greedy algorithm for the single-source/all-destinations

problem can be implemented to have complexity $O(n \log n + e)$. For sparse graphs, this implementation is superior to that of Program 6.8. \square

Example 6.5: Consider the eight-vertex digraph of Figure 6.27(a) with length-adjacency matrix as in Figure 6.27(b). Suppose that the source vertex is Boston. The values of dist and the vertex u selected in each iteration of the for loop of lines 7 to 14 (Program 6.8) are shown in Figure 6.28. We use ∞ to denote the value *LARGE*. Note that the algorithm terminates after only 6 iterations of the for loop. By the definition of dist , the distance of the last vertex, in this case Los Angeles, is correct, as the shortest path from Boston to Los Angeles can go through only the remaining six vertices. \square

6.4.2 Single Source/All Destinations: General Weights

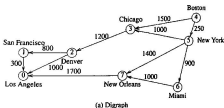
We now consider the general case when some or all of the edges of the directed graph G may have negative length. To see that function *ShortestPath* (Program 6.8) does not necessarily give the correct results on such graphs, consider the graph of Figure 6.29. Let $v = 0$ be the source vertex. Since $n = 3$, the loop of lines 7 to 14 is iterated just once: $u = 2$ in line 8, and no changes are made to dist . The function terminates with $\text{dist}[1] = 7$ and $\text{dist}[2] = 5$. The shortest path from 0 to 2 is 0, 1, 2. This path has length 2, which is less than the computed value of $\text{dist}[2]$.

When negative edge lengths are permitted, we require that the graph have no cycles of negative length. This is necessary so as to ensure that shortest paths consist of a finite number of edges. For example, consider the graph of Figure 6.30. The length of the shortest path from vertex 0 to vertex 2 is $-\infty$, as the length of the path

$$0, 1, 0, 1, 0, 1, \dots, 0, 1, 2$$

can be made arbitrarily small. This is so because of the presence of the cycle 0, 1, 0, which has a length of -1 .

When there are no cycles of negative length, there is a shortest path between any two vertices of an n -vertex graph that has at most $n - 1$ edges on it. To see this, observe that a path that has more than $n - 1$ edges must repeat at least one vertex and hence must contain a cycle. Elimination of the cycles from the path results in another path with the same source and destination. This path is cycle-free and has a length that is no more than that of the original path, as the length of the eliminated cycles was at least zero. We can use this observation on the maximum number of edges on a cycle-free shortest path to obtain an algorithm to determine a shortest path from a source vertex to all remaining vertices in the graph. As in the case of function *ShortestPath* (Program 6.8), we shall



(b) Length-adjacency matrix

	0	1	2	3	4	5	6	7
0	0							
1	300	0						
2	1000	800	0					
3			1200	0				
4				1500	0	250		
5				1000		0	900	1400
6							0	1000
7	1700							0

Figure 6.27: Digraph for Example 6.5

compute only the length, $\text{dist}[u]$, of the shortest path from the source vertex v to u . An exercise examines the extension needed to construct the shortest paths.

Let $\text{dist}^i[u]$ be the length of a shortest path from the source vertex v to vertex u under the constraint that the shortest path contains at most i edges. Then, $\text{dist}^1[u] = \text{length}[v][u]$, $0 \leq u < n$. As noted earlier, when there are no cycles of negative length, we can limit our search for shortest paths to paths with at most $n - 1$ edges. Hence, $\text{dist}^{n-1}[u]$ is the length of an unrestricted shortest path from v to u .

Our goal then is to compute $\text{dist}^{n-1}[u]$ for all u . This can be done using

Iteration	Vertex selected	Distance							
		LA	SP	DEN	CHI	BOST	NY	MIA	NO
		[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
Initial	---	∞	∞	∞	1500	0	250	∞	∞
1	5	∞	∞	∞	1250	0	250	1150	1650
2	6	∞	∞	∞	1250	0	250	1150	1650
3	3	∞	∞	1450	1250	0	250	1150	1650
4	7	3050	∞	1450	1250	0	250	1150	1650
5	2	3350	3250	1450	1250	0	250	1150	1650
6	1	3350	3250	1450	1250	0	250	1150	1650

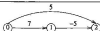
Figure 6.28: Action of *ShortestPath* on digraph of Figure 6.27

Figure 6.29: Directed graph with a negative-length edge



Figure 6.30: Directed graph with a cycle of negative length

the dynamic programming methodology. First, we make the following observation:

- (1) If the shortest path from v to u with at most k , $k > 1$, edges has no more than $k - 1$ edges, then $\text{dist}^k[u] = \text{dist}^{k-1}[u]$.
- (2) If the shortest path from v to u with at most k , $k > 1$, edges has exactly k edges, then it is comprised of a shortest path from v to some vertex j

followed by the edge $\langle j, u \rangle$. The path from v to j has $k-1$ edges, and its length is $\text{dist}^{k-1}[j]$. All vertices i such that the edge $\langle i, u \rangle$ is in the graph are candidates for j . Since we are interested in a shortest path, the i that minimizes $\text{dist}^{k-1}[i] + \text{length}[i][u]$ is the correct value for j .

These observations result in the following recurrence for dist :

$$\text{dist}^k[u] = \min\{\text{dist}^{k-1}[u], \min_i \{\text{dist}^{k-1}[i] + \text{length}[i][u]\}\}$$

This recurrence may be used to compute dist^k from dist^{k-1} , for $k = 2, 3, \dots, n-1$.

Example 6.6: Figure 6.31 gives a seven-vertex graph, together with the arrays dist^k , $k = 1, \dots, 6$. These arrays were computed using the equation just given. \square

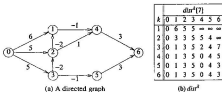


Figure 6.31: Shortest paths with negative edge lengths

An exercise shows that if we use the same memory location $\text{dist}[u]$ for $\text{dist}^k[u]$, $k = 1, \dots, n-1$, then the final value of $\text{dist}[u]$ is still $\text{dist}^{n-1}[u]$. Using this fact and the recurrence for dist shown above, we arrive at the algorithm of Program 6.9 to compute the length of the shortest path from vertex v to each other vertex of the graph. This algorithm is referred to as the Bellman and Ford algorithm.

Analysis of BellmanFord: Each iteration of the for loop of lines 4 to 7 takes $O(n^2)$ time if adjacency matrices are used and $O(e)$ time if adjacency lists are used. The overall complexity is $O(n^3)$ when adjacency matrices are used and

```

1 void MatrixWGraph::BellmanFord(const int n, const int v)
2 // Single source all destination shortest paths with negative edge lengths.
3 for (int i = 0; i < n; i++) dist[i] = length[v][i]; // initialize dist

4 for (int k = 2; k <= n-1; k++)
5   for (each u such that u != v and u has at least one incoming edge)
6     for (each <i, w> in the graph)
7       if (dist[u] > dist[i] + length[i][u]) dist[u] = dist[i] + length[i][u];
8 }

```

Program 6.9: Bellman and Ford algorithm to compute shortest paths

$O(n^3)$ when adjacency lists are used. The observed complexity of the shortest-path algorithm can be reduced by noting that if none of the dist values change on one iteration of the for loop of lines 4 to 7, then none will change on successive iterations. So, this loop may be rewritten to terminate either after $n-1$ iterations or after the first iteration in which no dist values are changed, whichever occurs first. Another possibility is to maintain a queue of vertices i whose dist value changed on the previous iteration of the for loop. These are the only values for i that need to be considered in line 6 during the next iteration. When a queue of these values is maintained, we can rewrite the loop of lines 4 to 7 so that on each iteration, a vertex i is removed from the queue, and the dist values of all vertices adjacent from i are updated as in line 7. Vertices whose dist value decreases as a result of this are added to the end of the queue unless they are already on it. The loop terminates when the queue becomes empty. \square

6.4.3 All-Pairs Shortest Paths

In the *all-pairs shortest-path problem*, we are to find the shortest paths between all pairs of vertices u and v , $u \neq v$. This problem can be solved as n independent single-source/all-destinations problems using each of the n vertices of G as a source vertex. If we use this approach on graphs with nonnegative edges, the total time taken would be $O(n^3)$ (or $O(n^2 \log n + ne)$ if Fibonacci heaps are used). On graphs with negative edges the run time will be $O(n^4)$ if adjacency matrices are used and $O(n^3e)$ if adjacency lists are used.

Using the dynamic programming approach to the design of algorithms, we can obtain a conceptually simpler algorithm that has complexity $O(n^3)$ and works even when G has edges with negative length. Like the Bellman and Ford algorithm, this algorithm requires that G have no cycles with negative length. This algorithm is faster for graphs with negative edges, as long as the graphs

have at least $c * n$ edges for some suitable constant c . Its observed run time is also less for dense graphs with nonnegative edge lengths. However, for sparse graphs with nonnegative edge lengths, using the single-source algorithm and Fibonacci heaps results in a faster algorithm for the all-pairs problem.

The graph G is represented by its length-adjacency matrix as described for function *ShortestPaths* (Program 6.8). Define $A^k[i][j]$ to be the length of the shortest path from i to j going through no intermediate vertex of index greater than k . Then, $A^{n-1}[i][j]$ will be the length of the shortest i -to- j path in G , since G contains no vertex with index greater than $n-1$. $A^{-1}[i][j]$ is just $\text{length}[i][j]$, since the only i -to- j paths allowed can have no intermediate vertices on them.

The basic idea in the all-pairs algorithm is to successively generate the matrices $A^{-1}, A^0, A^1, \dots, A^{n-1}$. If we have already generated A^{k-1} , then we may generate A^k by realizing that for any pair of vertices i and j , one of the following applies:

- (1) The shortest path from i to j going through no vertex with index greater than k does not go through the vertex with index k , so its length is $A^{k-1}[i][j]$.
- (2) The shortest path goes through vertex k . In this case, the path consists of a subpath from i to k and another one from k to j . These subpaths must be the shortest paths from i to k and from k to j going through no vertex with index greater than $k-1$, so their lengths are $A^{k-1}[i][k]$ and $A^{k-1}[k][j]$.

The preceding rules yield the following formulas for $A^k[i][j]$:

$$A^k[i][j] = \min\{A^{k-1}[i][j], A^{k-1}[i][k] + A^{k-1}[k][j]\}, k \geq 0$$

and

$$A^{-1}[i][j] = \text{length}[i][j]$$

The function *AllLengths* (Program 6.10) computes $A^{n-1}[i][j]$. The computation is done in place using the array a . The reason this computation can be carried out in place is that $A^k[i][k] = A^{k-1}[i][k]$ and $A^k[k][j] = A^{k-1}[k][j]$, so the in-place computation does not alter the outcome.

Analysis of AllLengths: This algorithm is especially easy to analyze because the looping is independent of the data in the matrix a . The total time for function *AllLengths* is $O(n^3)$. An exercise examines the extensions needed to obtain the (i, j) paths with these lengths. Some speed-up can be obtained by noticing that the innermost for loop needs be executed only when $a[i][k]$ is smaller than *LARGE*. \square

```

1 void MatrixW/Digraph::AllLengths(const int n)
2 { length [n] [n] is the adjacency matrix of a graph with n vertices.
3   a [i] [j] is the length of the shortest path between i and j.
4   for (int i = 0; i < n; i++)
5     for (int j = 0; j < n; j++)
6       a [i] [j] = length [i] [j]; // copy length into a
7   for (int k = 0; k < n; k++) // for a path with highest vertex index k
8     for (i = 0; i < n; i++) // for all possible pairs of vertices
9       for (int j = 0; j < n; j++)
10        if (a [i] [k] + a [k] [j] < a [i] [j]) a [i] [j] = a [i] [k] + a [k] [j];
11 }
```

Program 6.10: All-pairs shortest paths

Example 6.7: For the digraph of Figure 6.32(a), the initial a matrix, A^{-1} , plus its value after each of three iterations, A^0 , A^1 , and A^2 , is also given in Figure 6.32. \square

6.4.4 Transitive Closure

We end this section by studying a problem that is closely related to the all-pairs shortest-path problem. Assume that we have a graph G with unweighted edges. We want to determine if there is a path from i to j for all values of i and j . Two cases are of interest. The first case requires positive path lengths; the second requires only nonnegative path lengths. These cases are known as the *transitive closure* and *reflexive transitive closure* of a graph, respectively. We define them as follows:

Definition: The *transitive closure matrix*, denoted A^+ , of a graph, G , is a matrix such that $A^+[i][j] = 1$ if there is a path of length > 0 from i to j ; otherwise, $A^+[i][j] = 0$. \square

Definition: The *reflexive transitive closure matrix*, denoted A^* , of a graph, G , is a matrix such that $A^*[i][j] = 1$ if there is a path of length ≥ 0 from i to j ; otherwise, $A^*[i][j] = 0$. \square

Figure 6.33 shows A^+ and A^* for a digraph. Clearly, the only difference between A^+ and A^* is in the terms on the diagonal. $A^+[i][i] = 1$ iff there is a cycle of length > 1 containing vertex i , whereas $A^*[i][i]$ is always one, as there is



(a) Example digraph

A^{-1}	0	1	2
0	0	4	11
1	6	0	2
2	3	6	0

(b) A^{-1}

A^0	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0

(c) A^0

A^1	0	1	2
0	0	4	6
1	6	0	2
2	3	7	0

(d) A^1

A^2	0	1	2
0	0	4	6
1	5	0	2
2	3	7	0

(e) A^2

Figure 6.32: Example for all-pairs shortest-paths problem

a path of length 0 from i to i .

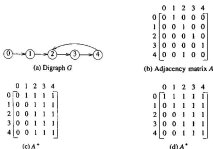
We can use function *AllLengths* (Program 6.10) to compute A^* . We begin with $\text{length}[i][j] = 1$ if $\langle i, j \rangle$ is an edge in G and $\text{length}[i][j] = \text{LARGE}$ if $\langle i, j \rangle$ is not in G . When *AllLengths* terminates, we can obtain A^* from the final matrix a by letting $A^*[i][j] = 1$ iff $a[i][j] < \text{LARGE}$. A^* can be obtained from A^* by setting all diagonal elements equal to 1. The total time is $O(n^3)$.

Some simplification is achieved by slightly modifying function *AllLengths*. In this modification, we make length and a arrays of type *bool*. Initially, $\text{length}[i][j] = \text{true}$ iff $\langle i, j \rangle$ is an edge of G . That is, length is the adjacency matrix of the graph. Line 10 of *AllLengths* is replaced by

```
 $a[i][j] = a[i][j] \vee (a[i][k] \ \&\& \ a[k][j])$ 
```

Upon termination of *AllLengths*, the final matrix a is A^* .

The transitive closure of an undirected graph G can be found more easily from its connected components. From the definition of a connected component, it follows that there is a path between every pair of vertices in the component

Figure 6.33: Graph G and its adjacency matrix A , A^* , and A^+

and there is no path in G between two vertices that are in different components. Hence, if A is the adjacency matrix of an undirected graph (i.e., A is symmetric) then its transitive closure A^* may be determined in $O(n^3)$ time by first determining the connected components of the graph. $A^*[i][j] = 1$ iff there is a path from vertex i to j . For every pair of distinct vertices in the same component, $A^*[i][j] = 1$. On the diagonal, $A^*[i][i] = 1$ iff the component containing i has at least two vertices.

EXERCISES

1. Let T be a tree with root v . The edges of T are undirected. Each edge in T has a nonnegative length. Write a C++ function to determine the length of the shortest paths from v to the remaining vertices of T . Your function should have complexity $O(n)$, where n is the number of vertices in T . Show that this is the case.

2. Let G be a directed, acyclic graph with n vertices. Assume that the vertices are numbered 0 through $n-1$ and that all edges are of the form $\langle i, j \rangle$, where $i < j$. Assume that the graph is available as a set of adjacency lists and that each edge has a length (which may be negative) associated with it. Write a C++ function to determine the length of the shortest paths from vertex 0 to the remaining vertices. The complexity of your algorithm should be $O(n + e)$, where e is the number of edges in the graph. Show that this is the case.
3. (a) Do the previous exercise, but this time find the length of the longest paths instead of the shortest paths.
(b) Extend your algorithm of (a) to determine a longest path from vertex 0 to each of the remaining vertices.
4. What is a suitable value for *LARGE* in the context of function *ShortestPath* (Program 6.8)? Provide this as a function of the largest edge length *maxl* and the number of vertices n .
5. Using the idea of *ShortestPath* (Program 6.8), write a C++ function to find a minimum-cost spanning tree whose worst-case time is $O(n^2)$.
6. Use *ShortestPath* (Program 6.8) to obtain, in nondecreasing order, the lengths of the shortest paths from vertex 0 to all remaining vertices in the digraph of Figure 6.34.

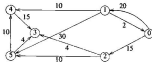


Figure 6.34: A digraph

7. Rewrite *ShortestPath* (Program 6.8) under the following assumptions:
 - (a) G is represented by its adjacency lists, where each node has three fields: *vertex*, *length*, and *link*. *length* is the length of the corresponding edge and n the number of vertices in G .
 - (b) Instead of S (the set of vertices to which the shortest paths have already been found), the set $T = V(G) - S$ is represented using a

linked list.

What can you say about the computing time of your new function relative to that of *ShortestPath*?

3. Modify *ShortestPath* (Program 6.8) so that it obtains the shortest paths, in addition to the lengths of these paths. What is the computing time of your modified function?
9. Using the directed graph of Figure 6.35, explain why *ShortestPath* will not work properly. What is the shortest path between vertices 0 and 6?

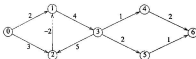


Figure 6.35: Directed graph on which *ShortestPath* does not work properly

10. Prove the correctness of function *BellmanFord* (Program 6.9). Note that this function does not faithfully implement the computation of the recurrence for dist^k . In fact, for $k < n - 1$, the dist values following iteration k of the for loop of lines 4 to 7 may not be dist^k .
11. Transform function *BellmanFord* into a complete C++ function. Assume that graphs are represented using adjacency lists in which each node has an additional field called *length* that gives the length of the edge represented by that node. As a result of this, there is no length-adjacency matrix. Generate some test graphs and test the correctness of your function.
12. Rewrite function *BellmanFord* so that the loop of lines 4 to 7 terminates either after $n - 1$ iterations or after the first iteration in which no dist values are changed, whichever occurs first.
13. Rewrite function *BellmanFord* by replacing the loop of lines 4 to 7 with code that uses a queue of vertices that may potentially result in a reduction of other dist vertices. This queue initially contains all vertices that are adjacent from the source vertex v . On each successive iteration of the new loop, a vertex i is removed from the queue (unless the queue is empty), and

the *dist* values to vertices adjacent from *i* are updated as in line 7 of Program 6.9. When the *dist* value of a vertex is reduced because of this, it is added to the queue unless it is already on the queue.

- (a) Prove that the new function produces the same results as the original one.
 - (b) Show that the complexity of the new function is no more than that of the original one.
14. Compare the run-time performance of the Bellman and Ford functions of the preceding two exercises and that of Program 6.9. For this, generate test graphs that will expose the relative performance of the three functions.
 15. Modify function *BellmanFord* so that it obtains the shortest paths, in addition to the lengths of these paths. What is the computing time of your function?
 16. What is a suitable value for *LARGE* in the context of function *AllLengths* (Program 6.10)? Provide this as a function of the largest edge length *maxL* and the number of vertices *n*.
 17. Modify function *AllLengths* (Program 6.10) so that it obtains a shortest path for all pairs of vertices. What is the computing time of your new function?
 18. Use function *AllLengths* to obtain the lengths of the shortest paths between all pairs of vertices in the graph of Figure 6.34. Does *AllLengths* give the right answers? Why?
 19. By considering the complete graph with *n* vertices, show that the maximum number of simple paths between two vertices is $O((n - 1)!)^2$.
 20. Show that $A^+ = A^+ \times A$, where matrix multiplication of the two matrices is defined as $a_{ij}^+ = \vee_{k=1}^n a_{ik} \wedge a_{kj}$. \vee is the logical or operation, and \wedge is the logical and operation.
 21. Obtain the matrices A^+ and A^* for the digraph of Figure 6.16.
 22. What is a suitable value for *LARGE* when *AllPaths* (Program 6.8) is used to compute the transitive closure of a directed graph? Provide this as a function of the number of vertices *n*.

6.5 ACTIVITY NETWORKS

6.5.1 Activity-on-Arrow (AOA) Networks

All but the simplest of projects can be subdivided into several subprojects called activities. The successful completion of these activities results in the completion of the entire project. A student working toward a degree in computer science

376 Graphs

must complete several courses successfully. The project in this case is to complete the major, and the activities are the individual courses that have to be taken. Figure 6.36 lists the courses needed for a computer science major at a hypothetical university. Some of these courses may be taken independently of others; other courses have prerequisites and can be taken only if all the prerequisites have already been taken. The data structures course cannot be started until certain programming and math courses have been completed. Thus, prerequisites define precedence relations between courses. The relationships defined may be more clearly represented using a directed graph in which the vertices represent courses and the directed edges represent prerequisites.

Definition: A directed graph G in which the vertices represent tasks or activities and the edges represent precedence relations between tasks is an *activity-on-arrow network* or *AOV network*. \square

Figure 6.36(b) is the AOV network corresponding to the courses of Figure 6.36(a). Each edge $\langle i, j \rangle$ implies that course i is a prerequisite of course j .

Definition: Vertex i in an AOV network G is a *predecessor* of vertex j iff there is a directed path from vertex i to vertex j . i is an *immediate predecessor* of j iff $\langle i, j \rangle$ is an edge in G . If i is a predecessor of j , then j is a *successor* of i . If i is an immediate predecessor of j , then j is an *immediate successor* of i . \square

C3 and C6 are immediate predecessors of C7. C9, C10, C12, and C13 are immediate successors of C7. C14 is a successor, but not an immediate successor, of C3.

Definition: A relation \cdot is *transitive* iff it is the case that for all triples i, j, k , $i \cdot j$ and $j \cdot k \Rightarrow i \cdot k$. A relation \cdot is *irreflexive* on a set S if for no element x in S is it the case that $x \cdot x$. A precedence relation that is both transitive and irreflexive is a *partial order*. \square

Notice that the precedence relation defined by course prerequisites is transitive. That is, if course i must be taken before course j (as i is a prerequisite of j), and if j must be taken before k , then i must be taken before k . This fact is not obvious from the AOV network. For example, $\langle C4, C3 \rangle$ and $\langle C3, C6 \rangle$ are edges in the AOV network of Figure 6.36(b). However, $\langle C4, C6 \rangle$ is not. Generally, AOV networks are incompletely specified, and the edges needed to make the precedence relation transitive are implied.

If the precedence relation defined by the edges of an AOV network is not irreflexive, then there is an activity that is a predecessor of itself and so must be completed before it can be started. This is clearly impossible. When there are

Course number	Course name	Prerequisites
C1	Programming I	None
C2	Discrete Mathematics	None
C3	Data Structures	C1, C2
C4	Calculus I	None
C5	Calculus II	C4
C6	Linear Algebra	C5
C7	Analysis of Algorithms	C3, C6
C8	Assembly Language	C3
C9	Operating Systems	C7, C8
C10	Programming Languages	C7
C11	Compiler Design	C10
C12	Artificial Intelligence	C7
C13	Computational Theory	C7
C14	Parallel Algorithms	C13
C15	Numerical Analysis	C5

(a) Courses needed for a computer science degree at a hypothetical university



(b) AOV network representing courses as vertices and prerequisites as edges

Figure 6.36: An activity-on-vertex (AOV) network

no inconsistencies of this type, the project is feasible. Given an AOV network, one of our concerns would be to determine whether or not the precedence relation defined by its edges is reflexive. This is identical to determining whether or not the network contains any directed cycles. A directed graph with no

directed cycles is an *acyclic* graph. Our algorithm to test an AOV network for feasibility will also generate a linear ordering, v_0, v_1, \dots, v_{n-1} , of the vertices (activities). This linear ordering will have the property that if vertex i is a predecessor of j in the network, then i precedes j in the linear ordering. A linear ordering with this property is called a *topological order*.

Definition: A *topological order* is a linear ordering of the vertices of a graph such that, for any two vertices i and j , if i is a predecessor of j in the network, then i precedes j in the linear ordering. \square

There are several possible topological orders for the network of Figure 6.36(b). Two of these are

C1, C2, C4, C5, C3, C6, C8, C7, C10, C13, C12, C14, C15, C11, C9

and

C4, C5, C2, C1, C6, C3, C8, C15, C7, C9, C10, C11, C12, C13, C14

If a student were taking just one course per term, then she or he would have to take them in topological order. If the AOV network represented the different tasks involved in assembling an automobile, then these tasks would be carried out in topological order on an assembly line. The algorithm to sort the tasks into topological order is straightforward and proceeds by listing a vertex in the network that has no predecessor. Then, this vertex together with all edges leading out from it is deleted from the network. These two steps are repeated until all vertices have been listed or all remaining vertices in the network have predecessors, and so none can be removed. In this case there is a cycle in the network, and the project is infeasible. The algorithm is stated more formally in Program 6.11.

```

1  Input the AOV network. Let  $n$  be the number of vertices.
2  for (int  $i = 0$ ;  $i < n$ ;  $i++$ )  $N$  output the vertices
3  {
4    if (every vertex has a predecessor) return;
5    if network has a cycle and is infeasible.
6    pick a vertex  $v$  that has no predecessors;
7    cout <<  $v$ ;
8    delete  $v$  and all edges leading out of  $v$  from the network;
9  }
```

Program 6.11: Design of a topological sorting algorithm

Example 6.8: Let us try out our topological sorting algorithm on the network of

Figure 6.37(a). The first vertex to be picked in line 6 is 0, as it is the only one with no predecessors. Vertex 0 and the edges $\langle 0, 1 \rangle$, $\langle 0, 2 \rangle$, and $\langle 0, 3 \rangle$ are deleted. In the resulting network (Figure 6.37(b)), vertices 1, 2, and 3 have no predecessor. Any of these can be the next vertex in the topological order. Assume that 3 is picked. Deletion of vertex 3 and the edges $\langle 3, 5 \rangle$ and $\langle 3, 4 \rangle$ results in the network of Figure 6.37(c). Either 1 or 2 may be picked next. Figure 6.37 shows the progress of the algorithm on the network. \square

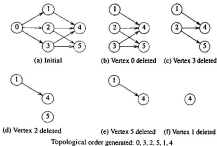


Figure 6.37: Action of Program 6.11 on an AOV network (shaded vertices represent candidates for deletion)

To obtain a complete algorithm that can be easily translated into a computer program, it is necessary to specify the data representation for the AOV network. The choice of a data representation, as always, depends on the functions you wish to perform. In this problem, the functions are

- (1) decide whether a vertex has any predecessors (line 6)
- (2) delete a vertex together with all its incident edges (line 8)

To perform the first function efficiently, we maintain a count of the number

of immediate predecessors each vertex has. The second function is easily implemented if the network is represented by its adjacency lists. Then the deletion of all edges leading out of vertex v can be carried out by decreasing the predecessor count of all vertices on its adjacency list. Whenever the count of a vertex drops to zero, that vertex can be placed onto a list of vertices with a zero count. Then the selection in line 6 just requires removal of a vertex from this list.

As a result of the preceding analysis, we represent the AOV network using adjacency lists. We assume that the adjacency representation defines a data member `count`, which is of type `int*` and that `count[i]` has been initialized to the in-degree of vertex i , $0 \leq i < n$. This can be done easily at the time of input. When edge $\langle i, j \rangle$ is input, the count of vertex j is incremented by 1. Figure 6.38(a) shows the internal representation of the network of Figure 6.37(a).

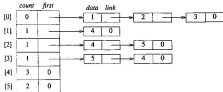


Figure 6.38: Internal representation used by topological sorting algorithm

Inserting these details into Program 6.11, we obtain the C++ function `TopologicalOrder` (Program 6.12). The list of vertices with zero count is maintained as a custom stack. A queue could have been used instead, but a stack is slightly simpler. The stack is linked through the `count` field of the header nodes, since this field is of no use after a vertex's count has become zero.

Analysis of `TopologicalOrder`: As a result of a judicious choice of data structures, the algorithm is very efficient. For a network with n vertices and e edges, the loop of lines 4 and 5 takes $O(n)$ time; lines 6 to 10 take $O(e)$ time over the entire algorithm; and the while loop of lines 11 to 15 takes time $O(d_i)$ for each

```

1 void LinkedDigraph::TopologicalOrder()
2 {// The  $n$  vertices of a network are listed in topological order.
3   int top = -1;
4   for (int i = 0; i < n; i++) // create a linked stack of vertices with
5     if (count[i] == 0) { count[i] = top; top = i; } // no predecessors

6   for (i = 0; i < n; i++)
7     if (top == -1) throw " Network has a cycle.";
8     int j = top; top = count[top]; // unstack a vertex
9     cout << j << endl;
10    Chain <int>; ChainIterator ji = adj[Edges[j]].begin();
11    while (ji) { // decrease the count of the successor vertices of j
12      count[*ji]--;
13      if (count[*ji] == 0) { count[*ji] = top; top = *ji; } // add *ji to stack
14      ji++;
15    }
16 }

```

Program 6.12: Topological order

vertex i , where d_i is the out-degree of vertex i . Since this loop is encountered once for each vertex output, the total time for this part of the algorithm is $O(\sum_{i=0}^{n-1} d_i + n) = O(e + n)$. Hence, the asymptotic computing time of the function is $O(e + n)$. It is linear in the size of the problem! \square

6.5.2 Activity-on-Edge (AOE) Networks

An activity network closely related to the AOV network is the *activity-on-edge*, or *AOE network*. The tasks to be performed on a project are represented by directed edges. Vertices in the network represent events. Events signal the completion of certain activities. Activities represented by edges leaving a vertex cannot be started until the event at that vertex has occurred. An event occurs only when all activities entering it have been completed. Figure 6.39(a) is an AOE network for a hypothetical project with 11 tasks or activities: a_1, \dots, a_{11} . There are nine events: 0, 1, \dots , 8. The events 0 and 8 may be interpreted as "start project" and "finish project," respectively. Figure 6.39(b) gives interpretations for some of the nine events. The number associated with each activity is the time needed to perform that activity. Thus, activity a_1 requires 6 days, whereas a_{11} requires 4 days. Usually, these times are only estimates. Activities

a_1 , a_2 , and a_3 may be carried out concurrently after the start of the project. Activities a_4 , a_5 , and a_6 cannot be started until events 1, 2, and 3, respectively, occur. Activities a_7 and a_8 can be carried out concurrently after the occurrence of event 4 (i.e., after a_4 and a_5 have been completed). If additional ordering constraints are to be put on the activities, dummy activities whose time is zero may be introduced. Thus, if we desire that activities a_7 and a_8 not start until both events 4 and 5 have occurred, a dummy activity a_{12} represented by an edge $\langle 5, 4 \rangle$ may be introduced.



(a) Activity network of a hypothetical project

event	interpretation
0	start of project
1	completion of activity a_1
4	completion of activities a_4 and a_5
7	completion of activities a_8 and a_9
8	completion of project

(b) Interpretation of some of the events in the network of (a)

Figure 6.39: An AOE network

Activity networks of the AOE type have proved very useful in the performance evaluation of several types of projects. This evaluation includes determining such facts about the project as what is the least amount of time in which the project may be completed (assuming there are no cycles in the network), which activities should be speeded to reduce project length, and so on.

Since the activities in an AOE network can be carried out in parallel, the minimum time to complete the project is the length of the longest path from the

start vertex to the finish vertex (the length of a path is the sum of the times of activities on this path). A path of longest length is a *critical path*. The path 0, 1, 4, 6, 8 is a critical path in the network of Figure 6.39(a). The length of this critical path is 18. A network may have more than one critical path (the path 0, 1, 4, 7, 8 is also critical).

The *earliest time* that an event i can occur is the length of the longest path from the start vertex 0 to the vertex i . The earliest time that event v_4 can occur is 7. The earliest time an event can occur determines the *earliest start time* for all activities represented by edges leaving that vertex. Denote this time by $e(i)$ for activity a_i . For example, $e(7)=e(8)=7$.

For every activity a_i , we may also define the *latest time*, $l(i)$, that an activity may start without increasing the project duration (i.e., length of the longest path from start to finish). In Figure 6.39(a) we have $e(8)=3$ and $l(8)=8$, $e(8)=7$ and $l(8)=7$.

All activities for which $e(i)=l(i)$ are called *critical activities*. The difference $l(i)-e(i)$ is a measure of the criticality of an activity. It gives the time by which an activity may be delayed or slowed without increasing the total time needed to finish the project. If activity a_8 is slowed down to take 2 extra days, this will not affect the project finish time. Clearly, all activities on a critical path are strategic, and speeding up noncritical activities will not reduce the project duration.

The purpose of critical-path analysis is to identify critical activities so that resources may be concentrated on these activities in an attempt to reduce project finish time. Speeding a critical activity will not result in a reduced project length unless that activity is on all critical paths. In Figure 6.39(a) the activity a_{11} is critical, but speeding it up so that it takes only 3 days instead of 4 does not reduce the finish time to 17 days. This is so because there is another critical path (0, 1, 4, 6, 8) that does not contain this activity. The activities a_1 and a_4 are on all critical paths. Speeding a_1 by 2 days reduces the critical path length to 16 days. Critical-path methods have proved very valuable in evaluating project performance and identifying bottlenecks.

Critical-path analysis can also be carried out with AOV networks. The length of a path would now be the sum of the activity times of the vertices on that path. By analogy, for each activity or vertex we could define the quantities $e(i)$ and $l(i)$. Since the activity times are only estimates, it is necessary to re-evaluate the project during several stages of its completion as more accurate estimates of activity times become available. These changes in activity times could make previously noncritical activities critical, and vice versa.

Before ending our discussion on activity networks, let us design an algorithm to calculate $e(i)$ and $l(i)$ for all activities in an AOE network. Once these quantities are known, then the critical activities may easily be identified. Deferring all noncritical activities from the AOE network, all critical paths may be

found by just generating all paths from the start-to-finish vertex (all such paths will include only critical activities and so must be critical, and since no noncritical activity can be on a critical path, the network with noncritical activities removed contains all critical paths present in the original network).

6.5.2.1 Calculation of Early Activity Times

When computing the early and late activity times, it is easiest first to obtain the earliest event time, $ee(j)$, and latest event time, $le(j)$, for all events, j , in the network. Thus if activity a_i is represented by edge $\langle k, l \rangle$, we can compute $e(i)$ and $l(i)$ from the following formulas:

$$e(i) = ee(k)$$

and

$$l(i) = le(l) - \text{duration of activity } a_i \quad (6.1)$$

The times $ee(j)$ and $le(j)$ are computed in two stages: a forward stage and a backward stage. During the forward stage we start with $ee(0) = 0$ and compute the remaining early start times, using the formula

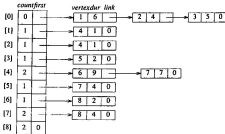
$$ee(j) = \max_{i \in P(j)} \{ ee(i) + \text{duration of } \langle i, j \rangle \} \quad (6.2)$$

where $P(j)$ is the set of all vertices adjacent to vertex j . If this computation is carried out in topological order, the early start times of all predecessors of j would have been computed prior to the computation of $ee(j)$. So, if we modify *TopologicalOrder* (Program 6.12) so that it returns the vertices in topological order (rather than outputs them in this order), then we may use this topological order and Eq. 6.2 to compute the early event times. To use Eq. 6.2, however, we must have easy access to the vertex set $P(j)$. Since the adjacency list representation does not provide easy access to $P(j)$, we make a more major modification to Program 6.12. We begin with the *ee* array initialized to zero and insert the line

ee["*j*"] = max(*ee*["*i*"], *ee*["*j*] + duration of $\langle j, i \rangle$];

between lines 12 and 13. This modification results in the evaluation of Eq. (6.2) in parallel with the generation of a topological order. *ee*["*j*"] is updated each time the *ee*["*i*"] of one of its predecessors is known (i.e., when *i* is ready for output).

To illustrate the working of the modified *TopologicalOrder* algorithm, let us try it out on the network of Figure 6.39(a). The adjacency lists for the network are shown in Figure 6.40(a). The order of nodes on these lists determines



(a) Adjacency lists for Figure 6.39(a)

ee	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	Stack
initial	0	0	0	0	0	0	0	0	0	[0]
output 0	0	6	4	5	0	0	0	0	0	[3, 2, 1]
output 3	0	6	4	5	0	7	0	0	0	[5, 2, 1]
output 5	0	6	4	5	0	7	0	11	0	[2, 1]
output 2	0	6	4	5	5	7	0	11	0	[1]
output 1	0	6	4	5	7	7	0	11	0	[4]
output 4	0	6	4	5	7	7	16	14	0	[7, 6]
output 7	0	6	4	5	7	7	16	14	18	[6]
output 6	0	6	4	5	7	7	16	14	18	[8]

(b) Computation of ee

Figure 6.40: Computing ee using modified *TopologicalOrder* (Program 6.12)

the order in which vertices will be considered by the algorithm. At the outset, the early start time for all vertices is 0, and the start vertex is the only one in the stack. When the adjacency list for this vertex is processed, the early start time of all vertices adjacent from 0 is updated. Since vertices 1, 2, and 3 are now in the stack, all their predecessors have been processed, and Eq. (6.2) has been evaluated for these three vertices. $ee[5]$ is the next one determined. When vertex 5 is being processed, $ee[7]$ is updated to 11. This, however, is not the true value for $ee[7]$, since Eq. (6.2) has not been evaluated over all predecessors of 7 (v_4 has not yet been considered). This does not matter, as 7 cannot get stacked until all its predecessors have been processed. $ee[4]$ is next updated to 5 and finally to 7. At this point $ee[4]$ has been determined, as all the predecessors of 4 have been examined. The values of $ee[6]$ and $ee[7]$ are next obtained. $ee[8]$ is ultimately determined to be 18, the length of a critical path. You may readily verify that when a vertex is put into the stack, its early time has been correctly computed. The insertion of the new statement does not change the asymptotic computing time; it remains $O(e + v)$.

6.5.2.2 Calculation of Late Activity Times

In the backward stage the values of $le[j]$ are computed using a function analogous to that used in the forward stage. We start with $le[n-1] = ee[n-1]$ and use the equation

$$le[j] = \min_{i \in S(j)} \{le[i] - \text{duration of } \langle j, i \rangle\} \quad (6.3)$$

where $S(j)$ is the set of vertices adjacent from vertex j . The initial values for $le[i]$ may be set to $ee[n-1]$. Basically, Eq. (6.3) says that if $\langle j, i \rangle$ is an activity and the latest start time for event i is $le[i]$, then event j must occur no later than $le[i] - \text{duration of } \langle j, i \rangle$. Before $le[j]$ can be computed for some event j , the latest event time for all successor events (i.e., events adjacent from j) must be computed. Once we have obtained the topological order and $ee[n-1]$ from the modified version of Program 6.12, we may compute the late event times in reverse topological order using the adjacency list of vertex j to access the vertices in $S(j)$. This computation is shown below for our example of Figure 6.39(a).

$$\begin{aligned} le[8] &= ee[8] = 18 \\ le[6] &= \min\{le[8] - 2\} = 16 \\ le[7] &= \min\{le[8] - 4\} = 14 \\ le[4] &= \min\{le[6] - 9, le[7] - 7\} = 7 \\ le[5] &= \min\{le[4] - 1\} = 6 \\ le[2] &= \min\{le[4] - 1\} = 6 \end{aligned}$$

$$le[5] = \min\{le[7] - 4\} = 10$$

$$le[3] = \min\{le[5] - 2\} = 8$$

$$le[0] = \min\{le[1] - 6, le[2] - 4, le[3] - 5\} = 0$$

If the forward stage has already been carried out and a topological ordering of the vertices obtained, then the values of $le[i]$ can be computed directly, using Eq. (6.3), by performing the computations in the reverse topological order. The topological order generated in Figure 6.40(b) is 0, 3, 5, 2, 1, 4, 7, 6, 8. We may compute the values of $le[i]$ in the order 8, 6, 7, 4, 1, 2, 5, 3, 0, as all successors of an event precede that event in this order. In practice, one would usually compute both ee and le . The procedure would then be to compute ee first, using algorithm *TopologicalOrder*, modified as discussed for the forward stage, and then to compute le directly from Eq. (6.3) in reverse topological order.

Using the values of ee (Figure 6.40) and of le (above), and Eq. (6.1), we may compute the early and late times $e(i)$ and $l(i)$ and the degree of criticality (also called slack) of each task. Figure 6.41 gives the values. The critical activities are $a_1, a_4, a_7, a_8, a_{10}$, and a_{11} . Deleting all noncritical activities from the network, we get the directed graph or critical network of Figure 6.42. All paths from 0 to 8 in this graph are critical paths, and there are no critical paths in the original network that are not paths in this graph.

activity	early time	late time	slack	critical
	e	l	$l - e$	$l - e = 0$
a_1	0	0	0	Yes
a_2	0	2	2	No
a_3	0	3	3	No
a_4	6	6	0	Yes
a_5	4	6	2	No
a_6	5	8	3	No
a_7	7	7	0	Yes
a_8	7	7	0	Yes
a_9	7	10	3	No
a_{10}	16	16	0	Yes
a_{11}	14	14	0	Yes

Figure 6.41: Early, late, and criticality values

As a final remark on activity networks, we note that the function *TopologicalOrder* detects only directed cycles in the network. There may be other flaws, such as vertices not reachable from the start vertex (Figure 6.43). When a

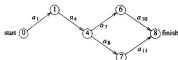


Figure 6.42: Graph obtained after deleting all noncritical activities

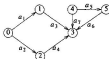


Figure 6.43: AOE network with some unreachable activities

critical-path analysis is carried out on such networks, there will be several vertices with $et(i) = 0$. Since all activity times are assumed > 0 , only the start vertex can have $et(i) = 0$. Hence, critical-path analysis can also be used to detect this kind of fault in project planning.

EXERCISES

- Does the following set of precedence relations ($<$) define a partial order on the elements 0 thru 4? Why?

$$0 < 1; 1 < 3; 1 < 2; 2 < 3; 2 < 4; 4 < 0$$

2. (a) For the AOE network of Figure 6.44 obtain the early, $e()$, and late, $l()$, start times for each activity. Use the forward-backward approach.
- (b) What is the earliest time the project can finish?
- (c) Which activities are critical?
- (d) Is there any single activity whose speed-up would result in a reduction of the project length?



Figure 6.44: An AOE network

3. Define a critical AOE network to be an AOE network in which all activities are critical. Let G be the undirected graph obtained by removing the directions and weights from the edges of the network.
 - (a) Show that the project length can be decreased by speeding up exactly one activity if there is an edge in G that lies on every path from the start vertex to the finish vertex. Such an edge is called a bridge. Deletion of a bridge from a connected graph separates the graph into two connected components.
 - (b) Write an $O(n + e)$ function using adjacency lists to determine whether the connected graph G has a bridge. If G has a bridge, your function should output one such bridge.
4. Write a C++ program that inputs an AOE network and outputs the following:
 - (a) A table of all events together with their earliest and latest times.
 - (b) A table of all activities together with their early and late times. This table should also list the slack for each activity and identify all critical activities (see Figure 6.41).

390 Graphs

- (c) The critical network.
 - (d) Whether or not the project length can be reduced by spending a single activity. If so, then by how much?
5. Define an iterator class *TopoIterator* for iterating through the vertices of a directed acyclic graph in topological order.

6.6 REFERENCES AND SELECTED READINGS

Euler's original paper on the Königsberg bridge problem makes interesting reading. This paper has been reprinted in: "Leonhard Euler and the Königsberg bridges," *Scientific American*, 189:1, 1953, pp. 66-70.

The biconnected-component algorithm is due to Robert Tarjan. This, together with a linear-time algorithm to find the strongly connected components of a directed graph, appears in the paper "Depth-first search and linear graph algorithms," by R. Tarjan, *SIAM Journal on Computing*, 1:2, 1972, pp. 146-159.

Prim's minimum-cost spanning tree algorithm was actually first proposed by Jarník in 1930 and rediscovered by Prim in 1957. Since virtually all references to this algorithm give credit to Prim, we continue to refer to it as Prim's algorithm. Similarly, the algorithm we refer to as Sollin's algorithm was first proposed by Borůvka in 1926 and rediscovered by Sollin several years later. For an interesting discussion of the history of the minimum spanning tree problem, see "On the history of the minimum spanning tree problem," by R. Graham and P. Hell, *Annals of the History of Computing*, 7:1, 1985, pp. 43-57.

Further algorithms on graphs may be found in *Graphs: Theory and applications*, by R. Thulasiraman and M. Swamy, Wiley Interscience, 1992.

6.7 ADDITIONAL EXERCISES

1. Program 6.13 was obtained by Stephen Barnard to find an Eulerian walk in a connected, undirected graph that has no vertices with odd degree.
- (a) Show that if G is represented by its adjacency multilists additions to path take $O(1)$ time, then function *Euler* works in time $O(n + e)$.
 - (b) Prove by induction on the number of edges in G that this algorithm does obtain an Eulerian walk for all graphs G having such a walk. The initial call to *Euler* can be made with any vertex v .
 - (c) At termination, what has to be done to determine whether or not G has an Eulerian walk?

```

virtual Path Graph::Euler(int v)
1 {
2     Path path = [ ];
3     for ((all vertices w adjacent to v) && (edge (v,w) not yet used)) {
4         mark edge (v,w) as used;
5         path = [(v,w)] ∪ euler(w) ∪ path;
6     }
7     return path;
8 }

```

Program 6.13: Finding an Eulerian walk

2. A bipartite graph $G = (V, E)$ is an undirected graph whose vertices can be partitioned into two disjoint sets, A and $B = V - A$, with the following properties: (1) No two vertices in A are adjacent in G , and (2) no two vertices in B are adjacent in G . The graph G_4 of Figure 6.3 is bipartite. A possible partitioning of V is $A = \{0, 3, 4, 6\}$ and $B = \{1, 2, 5, 7\}$. Write an algorithm to determine whether a graph G is bipartite. If G is bipartite your algorithm should obtain a partitioning of the vertices into two disjoint sets, A and B , satisfying properties (1) and (2) above. Show that if G is represented by its adjacency lists, then this algorithm can be made to work in time $O(n + e)$, where $n = |V|$ and $e = |E|$.
3. Show that every tree is a bipartite graph.
4. Prove that a graph G is bipartite iff it contains no cycles of odd length.
5. The radius of a tree is the maximum distance from the root to a leaf. Given a connected, undirected graph, write a function to find a spanning tree of minimum radius. (Hint: Use breadth-first search.) Prove that your algorithm is correct.
6. The diameter of a tree is the maximum distance between any two vertices. Given a connected, undirected graph, write an algorithm for finding a spanning tree of minimum diameter. Prove the correctness of your algorithm.
7. Let $G(n \parallel u)$ be a wiring grid. $G(i \parallel j) > 0$ represents a grid position that is blocked; $G(i \parallel j) = 0$ represents an unblocked position. Assume that positions $[a \parallel b]$ and $[c \parallel d]$ are blocked positions. A path from $[a \parallel b]$ to $[b \parallel c]$ is a sequence of grid positions such that
 - (a) $[a \parallel b]$ and $[c \parallel d]$ are, respectively, the first and last positions on the path
 - (b) successive positions of the sequence are vertically or horizontally

adjacent in the grid

- (c) all positions of the sequence other than the first and last are unblocked positions.

The length of a path is the number of grid positions on the path. We wish to connect positions $[a][b]$ and $[c][d]$ by a wire of shortest length. The wire path is a shortest grid path between these two vertices. Lee's algorithm for this works in the following steps:

- [Forward step] Start a breadth-first search from position $[a][b]$, labeling unblocked positions by their shortest distance from $[a][b]$. To avoid conflicts with existing labels, use negative labels. The labeling stops when the position $[c][d]$ is reached.
- [Backtrace] Use the labels of (a) to label the shortest path between $[a][b]$ and $[c][d]$, using the unique label $w > 0$ for the wire. For this, start at position $[c][d]$.
- [Clean-up] Change the remaining negative labels to 0.

Write algorithms for each of the three steps of Lee's algorithm. What is the complexity of each step?

8. Another way to represent a graph is by its incidence matrix, INC. There is one row for each vertex and one column for each edge. Then $\text{INC}[i][j] = 1$ if edge j is incident to vertex i . The incidence matrix for the graph of Figure 6.17(a) is given in Figure 6.45.

	0	1	2	3	4	5	6	7	8	9
0	1	1	0	0	0	0	0	0	0	0
1	1	0	1	1	0	0	0	0	0	0
2	0	1	0	0	1	1	0	0	0	0
3	0	0	1	0	0	0	1	0	0	0
4	0	0	0	1	0	0	0	1	0	0
5	0	0	0	0	1	0	0	0	1	0
6	0	0	0	0	0	1	0	0	0	1
7	0	0	0	0	0	0	1	1	1	1

Figure 6.45: Incidence matrix of graph of Figure 6.17(a)

The edges of Figure 6.17(a) have been numbered from left to right and top to bottom. Rewrite function *DFS* (Program 6.1) so that it works on a graph represented by its incidence matrix.

9. If ADJ is the adjacency matrix of a graph $G = (V, E)$, and INC is the incidence matrix, under what conditions will $ADJ = INC \times INC^T - I$, where INC^T is the transpose of matrix INC ? I is the identity matrix, and the matrix product $C = A \times B$, where all matrices are $n \times n$, is defined as $c_{ij} = \vee_{k=1}^n a_{ik} \wedge b_{kj}$. \vee is the \vee operation, and \wedge is the $\&\&$ operation.
10. An edge (u, v) of a connected, undirected graph G is a *bridge* iff its deletion from G results in a graph that is not connected. In the graph of Figure 6.20, the edges $(0, 1)$, $(3, 5)$, $(7, 8)$, and $(7, 9)$ are bridges. Write an algorithm that runs in $O(n + e)$ time to find the bridges of G . n and e are, respectively, the number of vertices and edges of G . (Hint: Use the ideas in function `isconnected` (Program 6.5).)
11. **[Programming Project]** Write a set of C++ classes for manipulating graphs. Such a collection should allow input and output of arbitrary graphs, determining connected components, spanning trees, minimum-cost spanning trees, biconnected components, shortest paths, and so on. You should include at least the classes of Figure 6.13.

CHAPTER 7

Sorting

7.1 MOTIVATION

In this chapter, we use the term *list* to mean a collection of records, each record having one or more fields. The fields used to distinguish among the records are known as *keys*. Since the same list may be used for several different applications, the key fields for record identification depend on the particular application. For instance, we may regard a telephone directory as a list, each record having three fields: name, address, and phone number. The key is usually the person's name. However, we may wish to locate the record corresponding to a given number, in which case the phone number field would be the key. In yet another application we may desire the phone number at a particular address, so the address field could also be the key.

One way to search for a record with the specified key is to examine the list of records in left-to-right or right-to-left order. Such a search is known as a *sequential search*. Program 7.1 gives a sequential search function that examines the records in left-to-right order. We assume that the relational operators ($<$, $>$, $=$, etc.) have been overloaded so that a comparison between a record of type E

and a key k of type K is done by comparing the record key with k .

```

template <class E, class K>
int SeqSearch (E *a, const int n, const K& k)
// Search a[1:n] from left to right. Return least i such that
// the key of a[i] equals k. If there is no such i, return 0.
    int i;
    for (i = 1; i <= n && a[i] != k; i++);
    if (i > n) return 0;
    return i;
}

```

Program 7.1: Sequential search

If no record in $a[1:n]$ has key value k , the search is *unsuccessful*. Program 7.1 makes n key comparisons when the search is unsuccessful. The number of key comparisons made in the case of a successful search depends on the position of the search key in the array a . If all keys are distinct and the key of $a[i]$ is being searched for, then i key comparisons are made. The average number of comparisons for a successful search is, therefore,

$$\left(\sum_{1 \leq i \leq n} i \right) / n = (n+1)/2.$$

It is possible to do much better than this when looking up phone numbers. The fact that the entries in the list (i.e., the telephone directory) are in lexicographic order (on the name key) enables one to look up a number while examining only a very few entries in the list. Binary search (see Chapter 1) is one of the better-known methods for searching an ordered, sequential list. A binary search takes only $O(\log n)$ time to search a list with n records. This is considerably better than the $O(n)$ time required by a sequential search. We note that when a sequential search is performed on an ordered list, the conditional of the for loop of SeqSearch can be changed to $i \leq n \ \&\& \ a[i] < k$. This change must be accompanied by a change of the conditional $i > n$ to $i > n \ \&\& \ a[i] != k$. These changes improve the performance of Program 7.1 for unsuccessful searches.

Getting back to our example of the telephone directory, we notice that neither a sequential nor a binary search strategy corresponds to the search method actually employed by humans. If we are looking for a name that begins with the letter W , we start the search toward the end of the directory rather than at the middle. A search method based on this interpolation scheme would begin by comparing k with $a[i]$, where $i = ((k - a[1].key)/a[n].key - a[1].key) * n$, and $a[1].key$ and $a[n].key$ are the smallest and largest keys in the list. An

interpolation search can be used only when the list is ordered. The behavior of such a search depends on the distribution of the keys in the list.

We have seen that something is to be gained by maintaining the list in an ordered manner if the list is to be searched repeatedly. Let us now look at another example in which the use of ordered lists greatly reduces the computational effort. The problem we are now concerned with is that of comparing two lists of records containing data that are essentially the same but have been obtained from two different sources. Such a problem could arise, for instance, in the case of the United States Internal Revenue Service (IRS), which might receive millions of forms from various employers stating how much they paid their employees and then another set of forms from individual employees stating how much they received. So we have two lists of records, and we wish to verify that there is no discrepancy between the two. Since the forms arrive at the IRS in a random order, we may assume a random arrangement of the records in the lists. The keys here are the social security numbers of the employees.

Let l_1 be the employer list and l_2 the employee list. Let $l_1[i].key$ and $l_2[j].key$, respectively, denote the key of the i th record in l_1 and l_2 . We make the following assumptions about the required verifications:

- (1) If there is no record in the employee list corresponding to a key in the employer list, a message is to be sent to the employee.
- (2) If the reverse is true, then a message is to be sent to the employer.
- (3) If there is a discrepancy between two records with the same key, a message to this effect is to be output.

Function *Verify1* (Program 7.2) solves the verification problem by directly comparing the two unsorted lists. The data type of the records in the list is *Element* and we assume that the relational operators have been overloaded so that a comparison between records is made by comparing their keys and that the output operator (`<<`) has been overloaded to output the record's key. The function *compare* returns *true* iff the two input records are identical in all fields. The complexity of *Verify1* is $O(mn)$, where n and m are, respectively, the number of records in the employer and employee lists. On the other hand, if we first sort the two lists and then do the comparison, we can carry out the verification task in time $O(t_{\text{sort}}(n) + t_{\text{sort}}(m) + n + m)$, where $t_{\text{sort}}(n)$ is the time needed to sort a list of n records. As we shall see, it is possible to sort n records in $O(n \log n)$ time, so the computing time becomes $O(\max\{n \log n, m \log m\})$. Function *Verify2* (Program 7.3) achieves this time.

We have seen two important uses of sorting: (1) as an aid in searching and (2) as a means for matching entries in lists. Sorting also finds application in the solution of many other more complex problems from areas such as optimization, graph theory and job scheduling. Consequently, the problem of sorting has great

```

void Verify1(Element *l1, Element *l2, const int n, const int m)
// Compare two unordered lists l1 and l2 of size n and m, respectively.
    bool *marked = new bool [m + 1];
    fill(marked + 1, marked + m + 1, false);

    for (i = 1; i <= n; i++)
    {
        int j = SeqSearch (l2, m, l1[i]);
        if (j == 0) cout << l1[i] << " not in l2 " << endl; // satisfies (1)
        else
        {
            if (!Compare (l1[i], l2[j]) // satisfies (3)
                cout << "Discrepancy in " << l1[i] << endl;
            marked [j] = true; // mark l2[j] as being seen
        }
    }
    for (i = 1; i <= m; i++)
        if (! marked [i]) cout << l2[i] << " not in l1. " << endl; // satisfies (2)
    delete [ ] marked ;
}

```

Program 7.2: Verifying two lists using a sequential search

relevance in the study of computing. Unfortunately, no one sorting method is the best for all applications. We shall therefore study several methods, indicating when one is superior to the others.

First let us formally state the problem we are about to consider. We are given a list of records (R_1, R_2, \dots, R_n) . Each record, R_i , has key value K_i . In addition, we assume an ordering relation ($<$) on the keys so that for any two key values x and y , $x = y$ or $x < y$ or $y < x$. The ordering relation ($<$) is assumed to be transitive (i.e., for any three values x, y , and z , $x < y$ and $y < z$ implies $x < z$). The sorting problem then is that of finding a permutation, σ , such that $K_{\sigma(1)} \leq K_{\sigma(2)} \leq \dots \leq K_{\sigma(n)}$. The desired ordering is $(R_{\sigma(1)}, R_{\sigma(2)}, \dots, R_{\sigma(n)})$.

Note that when the list has several key values that are identical, the permutation, σ , is not unique. We shall distinguish one permutation, σ_1 , from the others that also order the list. Let σ_1 be the permutation with the following properties:

- (1) $K_{\sigma_1(i)} \leq K_{\sigma_1(i+1)}$, $1 \leq i \leq n - 1$.
- (2) If $i < j$ and $K_i = K_j$ in the input list, then R_i precedes R_j in the sorted list.

```

void Verify2(Element *I1, Element *I2, const int n, const int m)
// Same task as Verify1. However, this time we first sort I1 and I2.
    Sort(I1, n); // sort into increasing order of key
    Sort(I2, m);
    int i = 1, j = 1;
    while ((i <= n) && (j <= m))
        if (I1[i] < I2[j])
        {
            cout << I1[i] << " not in I2" << endl;
            i++;
        }
        else if (I1[i] > I2[j])
        {
            cout << I2[j] << " not in I1" << endl;
            j++;
        }
        else
        { // equal keys
            if (!Compare(I1[i], I2[j]))
                cout << "Discrepancy in " << I1[i] << endl;
            i++; j++;
        }
    if (i <= n) OutputRan(I1, i, n, 1); // output records i through n of I1
    else if (j <= m) OutputRan(I2, j, m, 2); // output records j through m of I2
}

```

Program 7.3c Fast verification of two lists

A sorting method that generates the permutation σ_s is *stable*.

We characterize sorting methods into two broad categories: (1) internal methods (i.e., methods to be used when the list to be sorted is small enough so that the entire sort can be carried out in main memory) and (2) external methods (i.e., methods to be used on larger lists). The following internal sorting methods will be developed: Insertion Sort, Quick Sort, Merge Sort, Heap Sort, and Radix Sort. This development will be followed by a discussion of external sorting. Throughout, we assume that relational operators have been overloaded so that record comparison is done by comparing their keys.

7.2 INSERTION SORT

The basic step in this method is to insert a new record into a sorted sequence of i records in such a way that the resulting sequence of size $i + 1$ is also ordered. Function *Insert* (Program 7.4) accomplishes this insertion.

```

template <class T>
void Insert(const T& e, T *a, int i)
// Insert e into the ordered sequence a[1:i] such that the
// resulting sequence a[1:i+1] is also ordered.
// The array a must have space allocated for at least i + 2 elements.
{
    a[0] = e;
    while (e < a[i])
    {
        a[i+1] = a[i];
        i--;
    }
    a[i+1] = e;
}

```

Program 7.4: Insertion into a sorted list

The use of $a[0]$ enables us to simplify the while loop, avoiding a test for end of list (i.e., $i < 1$). In Insertion Sort, we begin with the ordered sequence $a[1]$ and successively insert the records $a[2], a[3], \dots, a[n]$. Since each insertion leaves the resultant sequence ordered, the list with n records can be ordered making $n - 1$ insertions. The details are given in function *InsertionSort* (Program 7.5).

Analysis of *InsertionSort*: In the worst case *Insert*(x, a, i) makes $i + 1$ comparisons before making the insertion. Hence the complexity of *Insert* is $O(i)$. *InsertionSort* invokes *Insert* for $i = j - 1 = 1, 2, \dots, n - 1$. So, the complexity of *InsertionSort* is

$$O\left(\sum_{i=1}^{n-1} (i+1)\right) = O(n^2).$$

We can also obtain an estimate of the computing time of Insertion Sort based on the relative disorder in the input list. Record R_i is *left out of order* (LOO) iff $R_i < \max_{1 \leq j < i} \{R_j\}$. The insertion step has to be carried out only for those records that are LOO. If k is the number of LOO records, the computing time is

400 Sorting

```
template <class T>
void InsertionSort(T *a, const int n)
{ // Sort a[1:n] into nondecreasing order.
  for (int j = 2; j <= n; j++) {
    T temp = a[j];
    Insert(temp, a, j-1);
  }
}
```

Program 7.5: Insertion Sort

$O(k+1)n) = O(kn)$. We can show that the average time for *InsertionSort* is $O(n^2)$ as well. \square

Example 7.1: Assume that $n = 5$ and the input key sequence is 5, 4, 3, 2, 1. After each insertion we have

j	[1]	[2]	[3]	[4]	[5]
—	5	4	3	2	1
2	4	5	3	2	1
3	3	4	5	2	1
4	2	3	4	5	1
5	1	2	3	4	5

For convenience, only the key field of each record is displayed, and the sorted part of the list is shown in bold. Since the input list is in reverse order, as each new record is inserted into the sorted part of the list, the entire sorted part is shifted right by one position. Thus, this input sequence exhibits the worst-case behavior of Insertion Sort. \square

Example 7.2: Assume that $n = 5$ and the input key sequence is 2, 3, 4, 5, 1. After each iteration we have

j	[1]	[2]	[3]	[4]	[5]
—	2	3	4	5	1
2	2	3	4	5	1
3	2	3	4	5	1
4	2	3	4	5	1
5	1	2	3	4	5

In this example, only record 5 is LOO, and the time for each $j = 2, 3$, and 4 is $O(1)$, whereas for $j = 5$ it is $O(n)$. \square

It should be fairly obvious that *InsertionSort* is stable. The fact that the computing time is $O(kn)$ makes this method very desirable in sorting sequences in which only a very few records are LOO (i.e., $k \ll n$). The simplicity of this scheme makes it about the fastest sorting method for small n (say, $n \leq 30$).

Variations

1. *Binary Insertion Sort*: We can reduce the number of comparisons made in an insertion sort by replacing the sequential searching technique used in *Insert* (Program 7.4) with binary search. The number of record moves remains unchanged.

2. *Linked Insertion Sort*: The elements of the list are represented as a linked list rather than as an array. The number of record moves becomes *zero* because only the link fields require adjustment. However, we must retain the sequential search used in *Insert*.

EXERCISES

1. Write the status of the list (12, 2, 16, 30, 8, 38, 4, 10, 20, 6, 18) at the end of each iteration of the for loop of *InsertionSort* (Program 7.5).
2. Write a C++ function that implements Binary Insertion Sort. What is the worst-case number of comparisons made by your sort function? What is the worst-case number of record moves made? How do these compare with the corresponding numbers for Program 7.5?
3. Write a C++ function that implements Linked Insertion Sort. What is the worst-case number of comparisons made by your sort function? What is the worst-case number of record moves made? How do these compare with the corresponding numbers for Program 7.5?

402 Sorting

7.3 QUICKSORT

We now turn our attention to a sorting scheme with very good average behavior. The Quick Sort scheme developed by C. A. R. Hoare has the best average behavior among the sorting methods we shall be studying. In Quick Sort, we select a pivot record from among the records to be sorted. Next, the records to be sorted are reordered so that the keys of records to the left of the pivot are less than or equal to that of the pivot and those of the records to the right of the pivot are greater than or equal to that of the pivot. Finally, the records to the left of the pivot and those to its right are sorted independently (using the Quick Sort method recursively).

Program 7.6 gives the resulting Quick Sort function. To sort $a[l:n]$, the function invocation is `QuickSort(a, l, n)`. Function `QuickSort` assumes that $a[n+1]$ has been set to have a key at least as large as the remaining keys.

```
template <class T>
void QuickSort(T *a, const int left, const int right)
{ // Sort a[left:right] into nondecreasing order.
  // a[left] is arbitrarily chosen as the pivot. Variables i and j
  // are used to partition the subarray so that at any time a[m] ≤ pivot, m < i,
  // and a[m] ≥ pivot, m > j. It is assumed that a[left] ≤ a[right + 1].
  if (left < right) {
    int i = left,
        j = right + 1,
        pivot = a[left];
    do {
      do i++; while (a[i] < pivot);
      do j--; while (a[j] > pivot);
      if (i < j) swap(a[i], a[j]);
    } while (i < j);
    swap(a[left], a[j]);

    QuickSort(a, left, j - 1);
    QuickSort(a, j + 1, right);
  }
}
```

Program 7.6: Quick Sort

Example 7.3: Suppose we are to sort a list of 10 records with keys (26, 5, 37, 1,

61, 11, 59, 15, 48, 19). Figure 7.1 gives the status of the list at each call of *QuickSort*. Square brackets indicate sublists yet to be sorted. □

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	left	right
[26	5	37	1	61	11	59	15	48	19]	1	10
[11	5	19	1	15]	26	[59	61	48	37]	1	5
[1	5]	11	[19	15]	26	[59	61	48	37]	1	2
1	5	11	[19	15]	26	[59	61	48	37]	4	5
1	5	11	15	19	26	[59	61	48	37]	7	10
1	5	11	15	19	26	[48	37]	59	[61]	7	8
1	5	11	15	19	26	37	48	59	[61]	10	10
1	5	11	15	19	26	37	48	59	61		

Figure 7.1: Quick Sort example

Analysis of *QuickSort*: The worst-case behavior of *QuickSort* is examined in Exercise 2 and shown to be $O(n^2)$. However, if we are lucky, then each time a record is correctly positioned, the sublist to its left will be of the same size as that to its right. This would leave us with the sorting of two sublists, each of size roughly $n/2$. The time required to position a record in a list of size n is $O(n)$. If $T(n)$ is the time taken to sort a list of n records, then when the list splits roughly into two equal parts each time a record is positioned correctly, we have

$$\begin{aligned}
 T(n) &\leq cn + 2T(n/2), \text{ for some constant } c \\
 &\leq cn + 2(cn/2 + 2T(n/4)) \\
 &\leq 2cn + 4T(n/4) \\
 &\vdots \\
 &\leq cn \log_2 n + nT(1) = O(n \log n)
 \end{aligned}$$

Lemma 7.1 shows that the average computing time for function *QuickSort* is $O(n \log n)$. Moreover, experimental results show that as far as average computing time is concerned, Quick Sort is the best of the internal sorting methods we shall be studying.

Lemma 7.1: Let $T_{avg}(n)$ be the expected time for function *QuickSort* to sort a

404 Sorting

list with n records. Then there exists a constant k such that $T_{\text{avg}}(n) \leq k \log_2 n$ for $n \geq 2$.

Proof: In the call to *QuickSort*(*list*, 1, n), the pivot gets placed at position j . This leaves us with the problem of sorting two sublists of size $j - 1$ and $n - j$. The expected time for this is $T_{\text{avg}}(j - 1) + T_{\text{avg}}(n - j)$. The remainder of the function clearly takes at most cn time for some constant c . Since j may take on any of the values 1 to n with equal probability, we have

$$T_{\text{avg}}(n) \leq cn + \frac{1}{n} \sum_{j=1}^n (T_{\text{avg}}(j - 1) + T_{\text{avg}}(n - j)) = cn + \frac{2}{n} \sum_{j=0}^{n-1} T_{\text{avg}}(j) \quad (7.1)$$

for $n \geq 2$. We may assume $T_{\text{avg}}(0) \leq b$ and $T_{\text{avg}}(1) \leq b$ for some constant b . We shall now show $T_{\text{avg}}(n) \leq k \log_2 n$ for $n \geq 2$ and $k = 2(b + c)$. The proof is by induction on n .

Induction base: For $n = 2$, Eq. (7.1) yields $T_{\text{avg}}(2) \leq 2c + 2b \leq k \log_2 2$.

Induction hypothesis: Assume $T_{\text{avg}}(n) \leq k \log_2 n$ for $1 \leq n < m$.

Induction step: From Eq. (7.1) and the induction hypothesis we have

$$T_{\text{avg}}(m) \leq cm + \frac{4b}{m} + \frac{2}{m} \sum_{j=1}^{m-1} T_{\text{avg}}(j) \leq cm + \frac{4b}{m} + \frac{2k}{m} \sum_{j=1}^{m-1} j \log_2 j \quad (7.2)$$

Since $j \log_2 j$ is an increasing function of j , Eq. (7.2) yields

$$\begin{aligned} T_{\text{avg}}(m) &\leq cm + \frac{4b}{m} + \frac{2k}{m} \int_1^m x \log_2 x \, dx = cm + \frac{4b}{m} + \frac{2k}{m} \left[\frac{m^2 \log_2 m}{2} - \frac{m^2}{4} \right] \\ &= cm + \frac{4b}{m} + k m \log_2 m - \frac{k m}{2} \leq k m \log_2 m, \text{ for } m \geq 2 \quad \square \end{aligned}$$

Unlike Insertion Sort, where the only additional space needed was for one record, Quick Sort needs stack space to implement the recursion. If the lists split evenly, as in the above analysis, the maximum recursion depth would be $\log n$, requiring a stack space of $O(\log n)$. The worst case occurs when the list is split into a left sublist of size $n - 1$ and a right sublist of size 0 at each level of recursion. In this case, the depth of recursion becomes n , requiring stack space of $O(n)$. The worst-case stack space can be reduced by a factor of 4 by realizing that right sublists of size less than 2 need not be stacked. An asymptotic reduction in stack space can be achieved by sorting smaller sublists first. In this case the additional stack space is at most $O(\log n)$.

Variation—Quick Sort using a median-of-three: Our version of Quick Sort always picked the key of the first record in the current sublist as the pivot. A better choice for this pivot is the median of the first, middle, and last keys in the current sublist. Thus, $\text{pivot} = \text{median} \{K_1, K_{(l+r)/2}, K_r\}$. For example, $\text{median}\{10, 5, 7\} = 7$ and $\text{median}\{10, 7, 7\} = 7$.

EXERCISES

1. Draw a figure similar to Figure 7.1 starting with the list (12, 2, 16, 30, 8, 28, 4, 10, 20, 6, 18).
2. (a) Show that *QuickSort* takes $O(n^2)$ time when the input list is already in sorted order.
(b) Show that the worst-case time complexity of *QuickSort* is $O(n^3)$.
(c) Why is $\text{list}[\text{left}] \leq \text{list}[\text{right} + 1]$ required in *QuickSort*?
3. (a) Write a nonrecursive version of *QuickSort* incorporating the median-of-three rule to determine the pivot key.
(b) Show that this function takes $O(n \log n)$ time on an already sorted list.
4. Show that if smaller sublists are sorted first, then the recursion in *QuickSort* can be simulated by a stack of depth $O(\log n)$.
5. Quick Sort is an unstable sorting method. Give an example of an input list in which the order of records with equal keys is not preserved.

7.4 HOW FAST CAN WE SORT?

Both of the sorting methods we have seen so far have a worst-case behavior of $O(n^2)$. It is natural at this point to ask the question, What is the best computing time for sorting that we can hope for? The theorem we shall prove shows that if we restrict our question to sorting algorithms in which the only operations permitted on keys are comparisons and interchanges, then $O(n \log n)$ is the best possible time.

The method we use is to consider a tree that describes the sorting process. Each vertex of the tree represents a key comparison, and the branches indicate the result. Such a tree is called a *decision tree*. A path through a decision tree represents a sequence of computations that an algorithm could produce.

Example 7.4: Let us look at the decision tree obtained for Insertion Sort working on a list with three records (Figure 7.2). The input sequence is R_1, R_2 , and R_3 , so the root of the tree is labeled {1, 2, 3}. Depending on the outcome of the comparison between keys K_1 and K_3 , this sequence may or may not change. If

$K_3 < K_1$, then the sequence becomes $[2, 1, 3]$; otherwise it stays $[1, 2, 3]$. The full tree resulting from these comparisons is given in Figure 7.2.

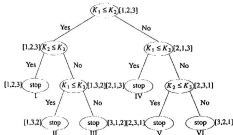


Figure 7.2: Decision tree for Insertion Sort

The leaf nodes are labeled I to VI. These are the only points at which the algorithm may terminate. Hence, only six permutations of the input sequence are obtainable from this algorithm. Since all six of these are different, and $3! = 6$, it follows that this algorithm has enough leaves to constitute a valid sorting algorithm for three records. The maximum depth of this tree is 3. Figure 7.3 gives six different orderings of the key values 7, 9, and 10, which show that all six permutations are possible. \square

Theorem 7.1: Any decision tree that sorts n distinct elements has a height of at least $\log_2(n!) + 1$.

Proof: When sorting n elements, there are $n!$ different possible results. Thus, every decision tree for sorting must have at least $n!$ leaves. But a decision tree is also a binary tree, which can have at most 2^{h-1} leaves if its height is h . Therefore, the height must be at least $\log_2 n! + 1$. \square

Corollary: Any algorithm that sorts only by comparisons must have a worst-

leaf	permutation	sample input key values that give the permutation
I	1 2 3	[7, 9, 10]
II	1 3 2	[7, 10, 9]
III	3 1 2	[9, 10, 7]
IV	2 1 3	[9, 7, 10]
V	2 3 1	[10, 7, 9]
VI	3 2 1	[10, 9, 7]

Figure 7.3: Sample input permutations

case computing time of $\tilde{O}(n \log n)$.

Proof: We must show that for every decision tree with $n!$ leaves, there is a path of length $c \log_2 n$, where c is a constant. By the theorem, there is a path of length $\log_2 n!$. Now

$$n! = n(n-1)(n-2) \cdots (3)(2)(1) \geq (n/2)^{n/2}.$$

So, $\log_2 n! \geq (n/2) \log_2(n/2) = \Omega(n \log n)$. \square

Using a similar argument and the fact that binary trees with 2^n leaves must have an average root-to-leaf path length of $\tilde{O}(n \log n)$, we can show that the average complexity of comparison-based sorting methods is $\tilde{O}(n \log n)$.

7.5 MERGE SORT

7.5.1 Merging

Before looking at the Merge Sort method to sort n records, let us see how one may merge two sorted lists to get a single sorted list. We shall examine two different algorithms. The first one, Program 7.7, is very simple and uses $\tilde{O}(n)$ additional space. The two lists to be merged are *inList*[$l:n$] and *outList*[$m+1:n$]. The resulting merged list is *mergedList*[$l:n$].

Analysis of Merge: At each iteration of the for loop, *iResult* increases by 1. The total increment in *iResult* is at most $n - l + 1$. Hence, the for loop is iterated

```

template <class T>
void Merge(T *initList, T *mergedList, const int l, const int m, const int n)
// initList[l:m] and initList[m + 1:n] are sorted lists. They are merged to obtain
// the sorted list mergedList[l:n].
    for (int i1 = l, iResult = l, i2 = m + 1; // i1, i2, and iResult are list positions
        i1 <= m && i2 <= n; // neither input list is exhausted
        iResult++)
        if (initList[i1] <= initList[i2])
        {
            mergedList[iResult] = initList[i1];
            i1++;
        }
        else
        {
            mergedList[iResult] = initList[i2];
            i2++;
        }
    // copy remaining records, if any, of first list
    copy(initList + i1, initList + m + 1, mergedList + iResult);

    // copy remaining records, if any, of second list
    copy(initList + i2, initList + n + 1, mergedList + iResult);
}

```

Program 7.7: Merging two sorted lists

at most $n - i + 1$ times. The copy statements copy at most $n - i + 1$ records. The total time is therefore $O(n - i + 1)$.

If each record has a size s , then the time is $O(s(n - i + 1))$. When s is greater than 1, we can use linked lists instead of arrays and obtain a new sorted linked list containing these $n - i + 1$ records. Now, we will not need the additional space for $n - i + 1$ records as needed in *Merge* for the array *mergedList*. Instead, space for $n - i + 1$ links is needed. The merge time becomes $O(n - i + 1)$ and is independent of s . Note that $n - i + 1$ is the number of records being merged. \square

7.5.2 Iterative Merge Sort

This version of Merge Sort begins by interpreting the input list as comprised of n sorted sublists, each of size 1. In the first merge pass, these sublists are merged by pairs to obtain $n/2$ sublists, each of size 2 (if n is odd, then one sublist is of size 1). In the second merge pass, these $n/2$ sublists are then merged by pairs to obtain $n/4$ sublists. Each merge pass reduces the number of sublists by half. Merge passes are continued until we are left with only one sublist. The example below illustrates the process.

Example 7.5: The input list is (26, 5, 77, 1, 61, 11, 59, 15, 48, 19). The tree of Figure 7.4 illustrates the sublists being merged at each pass. □

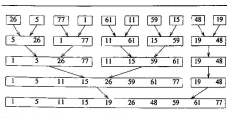


Figure 7.4: Merge tree

Since a Merge Sort consists of several merge passes, it is convenient first to write a function (Program 7.8) for a merge pass. Now the sort can be done by repeatedly invoking the merge-pass function as in Program 7.9.

Analysis of MergeSort: Function *MergeSort* makes several passes over the records being sorted. In the first pass, lists of size 1 are merged. In the second, the size of the lists being merged is 2. On the i th pass the lists being merged are of size 2^{i-1} . Consequently, a total of $\lceil \log_2 n \rceil$ passes are made over the data. Since two lists can be merged in linear time (function *Merge*), each pass of Merge Sort takes $O(n)$ time. The total computing time is $O(n \log n)$. □

```

template <class T>
void MergePass(T *initList, T *resultList, const int n, const int s)
// Adjacent pairs of sublists of size s are merged from
// initList to resultList. n is the number of records in initList.
    for (int i = 1; // i is first position in first of the sublists being merged
        i <= n - 2*s + 1; // enough elements for two sublists of length s?
        i += 2*s)
        Merge(initList, resultList, i, i + s - 1, i + 2 * s - 1);

// merge remaining list of size < 2 * s
if ((i + s - 1) < n) Merge(initList, resultList, i, i + s - 1, n);
else copy (initList + i, initList + n + 1, resultList + i);
}

```

Program 7.8: Merge pass

```

template <class T>
void MergeSort(T *a, const int n)
// Sort a[1:n] into nondecreasing order.
    T *tempList = new T[n + 1];
    // l is the length of the sublist currently being merged
    for (int l = 1; l <= n; l *= 2)
    {
        MergePass(a, tempList, n, l);
        l *= 2;
        MergePass(tempList, a, n, l); // interchange role of a and tempList
    }
    delete [] tempList;
}

```

Program 7.9: Merge Sort

You may verify that *MergeSort* is a stable sorting function.

7.5.3 Recursive Merge Sort

In the recursive formulation we divide the list to be sorted into two roughly equal parts called the left and the right sublists. These sublists are sorted recursively.

and the sorted sublists are merged.

Example 7.6: The input list (26, 5, 77, 1, 61, 11, 59, 15, 49, 19) is to be sorted using the recursive formulation of Merge Sort. If the sublist from *left* to *right* is currently to be sorted, then its two sublists are indexed from *left* to $\lfloor (left + right)/2 \rfloor$ and from $\lfloor (left + right)/2 \rfloor + 1$ to *right*. The sublist partitioning that takes place is described by the binary tree of Figure 7.5. Note that the sublists being merged are different from those being merged in MergeSort. □

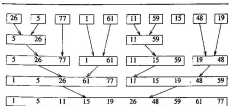


Figure 7.5: Sublist partitioning for Recursive Merge Sort

To eliminate the record copying that takes place when *Merge* (Program 7.7) is used to merge sorted sublists we associate an integer pointer with each record. For this purpose, we employ an integer array *link* $[1:n]$ such that *link* [*i*] gives the record that follows record *i* in the sorted sublist. In case *link* [*i*] = 0, there is no next record. With the addition of this array of links, record copying is replaced by link changes and the runtime of our sort function becomes independent of the size *s* of a record. Also the additional space required is $O(n)$. By comparison, the Iterative Merge Sort described earlier takes $O(n \log n)$ time and $O(nw)$ additional space. On the down side, the use of an array of links yields a sorted chain of records and we must have a follow up process to physically rearrange the records into the sorted order dictated by the final chain. We describe the algorithm for this physical rearrangement in Section 7.8.

We assume that initially *link* [*i*] = 0, $1 \leq i \leq n$. Thus, each record is initially in a chain containing only itself. Let *start1* and *start2* be pointers to two chains of records. The records on each chain are in nondecreasing order. Let

412 Sorting

ListMerge(*a*, *link*, *start1*, *start2*) be a function that merges two chains *start1* and *start2* in array *a* and returns the first position of the resulting chain that is linked in nondecreasing order of key values. The recursive version of Merge Sort is given by function *rdMergeSort* (Program 7.10). To sort the array *a*[1:*n*] this function is invoked as *rdMergeSort*(*a*, *link*, 1, *n*). The start of the chain ordered as described earlier is returned. Function *ListMerge* is given in Program 7.11.

```
template <class T>
int rdMergeSort(T* a, int* link, const int left, const int right)
// a[left:right] is to be sorted. link[i] is initially 0 for all i.
// rdMergeSort returns the index of the first element in the sorted chain.
    if (left >= right) return left;
    int mid = (left + right) / 2;
    return ListMerge(a, link,
                    rdMergeSort(a, link, left, mid), // sort left half
                    rdMergeSort(a, link, mid + 1, right)); // sort right half
}
```

Program 7.10: Recursive Merge Sort

Analysis of *rdMergeSort*: It is easy to see that the recursive Merge Sort is stable, and its computing time is $O(n \log n)$. \square

Variation—Natural Merge Sort: We may modify *MergeSort* to take into account the prevailing order within the input list. In this implementation we make an initial pass over the data to determine the sublists of records that are in order. Merge Sort then uses these initially ordered sublists for the remainder of the passes. Figure 7.6 shows Natural Merge Sort using the input sequence of Example 7.6.

EXERCISES

1. Write the status of the list (12, 2, 16, 30, 8, 28, 4, 10, 20, 6, 18) at the end of each phase of *MergeSort* (Program 7.9).
2. Suppose we use Program 7.12 to obtain a Merge Sort function. Is the resulting function a stable sort?
3. Prove that *MergeSort* is stable.

```

template <class T>
int LinkMerge(T* a, int* link, const int start1, const int start2)
// The sorted chains beginning at start1 and start2, respectively, are merged.
// link[0] is used as a temporary header. Return start of merged chain.
int iResult=0; // last record of result chain
for (int i1 = start1, i2 = start2; i1 && i2; )
    if (a[i1] <= a[i2]) {
        link[iResult] = i1;
        iResult = i1; i1 = link[i1];
    }
    else {
        link[iResult] = i2;
        iResult = i2; i2 = link[i2];
    }

// attach remaining records to result chain
if (i1 == 0) link[iResult] = i2;
else link[iResult] = i1;
return link[0];
}

```

Program 7.11: Merging sorted chains

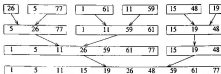


Figure 7.6: Natural Merge Sort

4.14 Sorting

4. Write an iterative Natural Merge Sort function using arrays as in function *MergeSort*. How much time does this function take on an initially sorted list? Note that *MergeSort* takes $O(n \log n)$ on such an input list. What is the worst-case computing time of the new function? How much additional space is needed?
5. Do the previous exercise using chains.

7.6 HEAP SORT

Although the Merge Sort scheme discussed in the previous section has a computing time of $O(n \log n)$, both in the worst case and as average behavior, it requires additional storage proportional to the number of records to be sorted. By using the $O(1)$ space merge algorithm, the space requirements can be reduced to $O(1)$. The resulting sort algorithm is, however, much slower than the original one. The sorting method we are about to study, Heap Sort, requires only a fixed amount of additional storage and at the same time has as its worst-case and average computing time $O(n \log n)$. Although Heap Sort is slightly slower than Merge Sort using $O(n)$ additional space, it is faster than Merge Sort using $O(1)$ additional space.

In Heap Sort, we utilize the max-heap structure introduced in Chapter 5. The deletion and insertion functions associated with max heaps directly yield an $O(n \log n)$ sorting method. The n records are first inserted into an initially empty max heap. Next, the records are extracted from the max heap one at a time. It is possible to create the max heap of n records faster than by inserting the records one by one into an initially empty heap. For this, we use the function *Adjust* (Program 7.13), which starts with a binary tree whose left and right subtrees are max heaps and rearranges records so that the entire binary tree is a max heap. The binary tree is embedded within an array using the standard mapping. If the depth of the tree is d , then the for loop is executed at most d times. Hence the computing time of *Adjust* is $O(d)$.

To sort the list, first we create a max heap by using *Adjust* repeatedly, as in the first for loop of function *HeapSort* (Program 7.14). Next, we swap the first and last records in the heap. Since the first record has the maximum key, the swap moves the record with maximum key into its correct position in the sorted array. We then decrement the heap size and readjust the heap. This swap, decrement heap size, readjust heap process is repeated $n - 1$ times to sort the entire array $a[1:n]$. Each repetition of the process is called a pass. For example, on the first pass, we place the record with the highest key in the n th position; on the second pass, we place the record with the second highest key in position $n - 1$; and on the i th pass, we place the record with the i th highest key in position $n - i + 1$. The invocation is *HeapSort*(a, n).

```

template <class T>
void Adjust(T *a, const int root, const int n)
// Adjust binary tree with root root to satisfy heap property. The left and right
// subtrees of root already satisfy the heap property. No node index is > n.
    T e = a[root];
    // find proper place for e
    for (int j = 2*root; j <= n; j *= 2) {
        if (j < n && a[j] < a[j+1]) j++; // j is max child of its parent
        if (e >= a[j]) break; // e may be inserted as parent of j
        a[j/2] = a[j]; // move jth record up the tree
    }
    a[j/2] = e;
}

```

Program 7.13: Adjusting a max heap

```

template <class T>
void HeapSort(T *a, const int n)
// Sort a[1:n] into nondecreasing order.
    for (int i = n/2; i >= 1; i--) // heapify
        Adjust(a, i, n);

    for (i = n-1; i >= 1; i--) // sort
    {
        swap(a[1], a[i+1]); // swap first and last of current heap
        Adjust(a, 1, i);    // heapify
    }
}

```

Program 7.14: Heap Sort

Example 7.7: The input list is (36, 5, 77, 1, 68, 11, 59, 15, 48, 19). If we interpret this list as a binary tree, we get the tree of Figure 7.7(a). Figure 7.7(b) depicts the max heap after the first for loop of *HeapSort*. Figure 7.8 shows the array of records following each of the first seven iterations of the second for loop. The portion of the array that still represents a max heap is shown as a binary tree; the sorted part of the array is shown as an array. □

Analysis of *HeapSort*: Suppose $2^{k-1} \leq n < 2^k$, so the tree has k levels and the number of nodes on level i is $\leq 2^{k-i}$. In the first for loop, *Adjust* (Program 7.13)

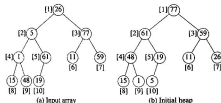


Figure 7.7: Array interpreted as a binary tree

is called once for each node that has a child. Hence, the time required for this loop is the sum, over each level, of the number of nodes on a level multiplied by the maximum-distance the node can move. This is no more than

$$\sum_{1 \leq i \leq k} 2^{k-i}(k-i) = \sum_{1 \leq i \leq k-1} 2^{k-i+1} i \leq n \sum_{1 \leq i \leq k-1} i/2^i < 2n = O(n)$$

In the next for loop, $n-1$ applications of *Adjust* are made with maximum rec-depth $k = \lceil \log_2(n+1) \rceil$ and *swap* is invoked $n-1$ times. Hence, the computing time for this loop is $O(n \log n)$. Consequently, the total computing time is $O(n \log n)$. Note that apart from some simple variables, the only additional space needed is space for one record to carry out the swap in the second for loop. \square

EXERCISES

- Write the status of the list (12, 2, 16, 30, 8, 38, 4, 10, 20, 6, 18) at the end of the first for loop as well as at the end of each iteration of the second for loop of *HeapSort* (Program 7.14).
- Heap Sort* is unstable. Give an example of an input list in which the order of records with equal keys is not preserved.

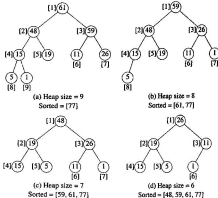


Figure 7.8: Heap Sort example (continued on next page)

7.7 SORTING ON SEVERAL KEYS

We now look at the problem of sorting records on several keys, K^1, K^2, \dots, K^r (K^1 is the most significant key and K^r the least). A list of records R_1, \dots, R_n is said to be sorted with respect to the keys K^1, K^2, \dots, K^r iff for every pair of records i and j , $i < j$ and $(K^1_i, \dots, K^r_i) \leq (K^1_j, \dots, K^r_j)$. The r -tuple (x_1, \dots, x_r) is less than or equal to the r -tuple (y_1, \dots, y_r) iff either $x_i = y_i$, $1 \leq i \leq j$, and $x_{j+1} < y_{j+1}$ for some $j < r$, or $x_i = y_i$, $1 \leq i \leq r$.

For example, the problem of sorting a deck of cards may be regarded as a sort on two keys, the suit and face values, with the following ordering relations:

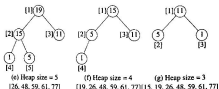


Figure 7.8: Heap Sort example

 K^1 [Suits]: ♠ < ♥ < ♦ < ♣

 K^2 [Face values]: 2 < 3 < 4 ··· < 10 < J < Q < K < A

A sorted deck of cards therefore has the following ordering:

$$2♠, \dots, A♠, \dots, 2♥, \dots, A♥$$

There are two popular ways to sort on multiple keys. In the first, we begin by sorting on the most significant key K^1 , obtaining several "piles" of records, each having the same value for K^1 . Then each of these piles is independently sorted on the key K^2 into "subpiles" such that all the records in the same subpile have the same values for K^1 and K^2 . The subpiles are then sorted on K^3 , and so on, and the piles are combined. Using this method on our card deck example, we would first sort the 52 cards into four piles, one for each of the suit values, then sort each pile on the face value. Then we would place the piles on top of each other to obtain the desired ordering.

A sort proceeding in this fashion is referred to as a most-significant-digit-first (MSD) sort. The second way, quite naturally, is to sort on the least significant digit first (LSD). An LSD sort would mean sorting the cards first into 13 piles corresponding to their face values (key K^2). Then, we would place the 3's on top of the 2's, ..., the kings on top of the queens, the aces on top of the kings; we would turn the deck upside down and sort on the suit (K^1) using a stable sorting method to obtain four piles, each ordered on K^2 ; and we would combine the piles to obtain the required ordering on the cards.

Comparing the two functions outlined here (MSD and LSD), we see that

LSD is simpler, as the piles and subpiles obtained do not have to be sorted independently (provided the sorting scheme used for sorting on the keys K^i , $1 \leq i < r$, is stable). This in turn implies less overhead.

The terms LSD and MSD specify only the order in which the keys are to be sorted. They do not specify how each key is to be sorted. When sorting a card deck manually, we generally use an MSD sort. The sorting on suit is done by a *bin sort* (i.e., four “bins” are set up, one for each suit value and the cards are placed into their corresponding bins). Next, the cards in each bin are sorted using an algorithm similar to Insertion Sort. However, there is another way to do this. First use a bin sort on the face value. To do this we need 13 bins, one for each distinct face value. Then collect all the cards together as described above and perform a bin sort on the suits using four bins. Note that a bin sort requires only $O(n)$ time if the spread in key values is $O(n)$.

LSD or MSD sorting can be used to sort even when the records have only one key. For this, we interpret the key as being composed of several subkeys. For example, if the keys are numeric, then each decimal digit may be regarded as a subkey. So, if the keys are in the range $0 \leq K \leq 999$, we can use either the LSD or MSD sorts for three keys (K^1 , K^2 , K^3), where K^1 is the digit in the hundreds place, K^2 the digit in the tens place, and K^3 the digit in the units place. Since $0 \leq K^i \leq 9$ for each key K^i , the sort on each key can be carried out using a bin sort with 10 bins.

In a *Radix Sort*, we decompose the sort key using some radix r . When r is 10, we get the decimal decomposition described above. When $r = 2$, we get binary decomposition of the keys. In a *Radix- r Sort*, the number of bins required is r .

Assume that the records to be sorted are R_1, \dots, R_n . The record keys are decomposed using a radix of r . This results in each key having d digits in the range 0 through $r - 1$. Thus, we shall need r bins. The records in each bin will be linked together into a chain with $f[i]$, $0 \leq i < r$, a pointer to the first record in bin i and $e[i]$, a pointer to the last record in bin i . These chains will operate as queues. Function *RadixSort* (Program 7.15) formally presents the LSD radix- r method.

Analysis of RadixSort: *RadixSort* makes d passes over the data, each pass taking $O(n + r)$ time. Hence, the total computing time is $O(d(n + r))$. The value of d will depend on the choice of the radix r and also on the largest key. Different choices of r will yield different computing times. \square

Example 7.8: Suppose we are to sort 10 numbers in the range [0, 999]. For this example, we use $r = 10$ (though other choices are possible). Hence, $d = 3$. The input list is linked and has the form given in Figure 7.9(a). The nodes are labeled R_1, \dots, R_{10} . Figure 7.9 shows the queues formed when sorting on each of the

```

template <class T>
int RadixSort(T *a, int *link, const int d, const int r, const int n)
{ // Sort a[1:n] using a d-digit radix-r sort. digit(a[i], j, r) returns the j-th radix-r
  // digit (from the left) of a[i]'s key. Each digit is in the range is [0, r).
  // Sorting within a digit is done using a bin sort.
    int e[r], f[r]; // queues front and end pointers
    // create initial chain of records starting at first
    int first = 1;
    for (int i = 1; i <= n; i++) link[i] = i + 1; // link into a chain
    link[n] = 0;

    for (j = d - 1; j >= 0; j--)
    { // sort on digit j
        fill(f, f + r, 0); // initialize bins to empty queues
        for (int current = first; current != link[current];)
        { // put records into queues/bins
            int k = digit(a[current], j, r);
            if (f[k] == 0) f[k] = current;
            else link[e[k]] = current;
            e[k] = current;
        }
        for (j = 0; !f[j]; j++); // find first nonempty queue/bin
        first = f[j];
        int last = e[j];
        for (int k = j + 1; k <= r; k++) // concatenate remaining queues
            if (f[k]) {
                link[last] = f[k];
                last = e[k];
            }
        link[last] = 0;
    }
    return first;
}

```

Program 7.15: LSD Radix Sort

digits, as well as the links after the queues have been collected from the 10 bins. □

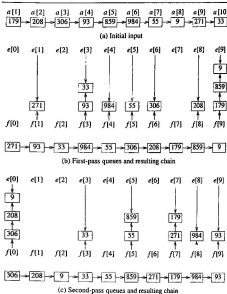


Figure 7.9: Radix Sort example (continued on next page)

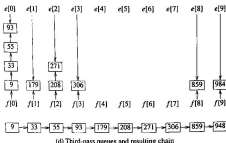


Figure 7.9: Radix Sort example

EXERCISES

1. Write the status of the list (12, 2, 16, 30, 8, 28, 4, 10, 20, 6, 18) at the end of each pass of *RadixSort* (Program 7.13). Use $r = 10$.
2. Under what conditions would an MSD Radix Sort be more efficient than an LSD Radix Sort?
3. Does *RadixSort* result in a stable sort when used to sort numbers as in Example 7.8?
4. Write a sort function to sort records R_1, \dots, R_n lexically on keys (K^1, \dots, K^r) for the case when the range of each key is much larger than n . In this case, the bin-sort scheme used in *RadixSort* to sort within each key becomes inefficient (why?). What scheme would you use to sort within a key if we desired a function with (a) good worst-case behavior, (b) good average behavior, (c) small n , say <15 ?
5. If we have n records with integer keys in the range $[0, n^2]$, then they may be sorted in $O(n \log n)$ time using Heap Sort or Merge Sort. Radix Sort on a single key (i.e., $d = 1$ and $r = n^2$) takes $O(n^2)$ time. Show how to interpret the keys as two subkeys so that Radix Sort will take only $O(n)$ time to

sort n records. (Hint: Each key, K_i , may be written as $K_i = K_i^1 n + K_i^2$ with K_i^1 and K_i^2 integers in the range $[0, n)$.)

6. Generalize the method of the previous exercise to the case of integer keys in the range $[0, n^p)$ obtaining an $O(pn)$ sorting method.
7. Experiment with *RadixSort* to see how it performs relative to the comparison-based sort methods discussed in earlier sections.

7.8 LIST AND TABLE SORTS

Apart from *Radix Sort* and *Recursive Merge Sort*, all the sorting methods we have looked at require excessive data movement. That is, as the result of a comparison, records may be physically moved. This tends to slow down the sorting process when records are large. When sorting lists with large records, it is necessary to modify the sorting methods so as to minimize data movement. Methods such as *Insertion Sort* and our *Iterative Merge Sort* can be modified to work with a linked list rather than a sequential list. In this case each record will require an additional link field. Instead of physically moving the record, we change its link field to reflect the change in the position of the record in the list. At the end of the sorting process, the records are linked together in the required order. In many applications (e.g., when we just want to sort lists and then output them record by record on some external media in the sorted order), this is sufficient. However, in some applications it is necessary to physically rearrange the records *in place* so that they are in the required order. Even in such cases, considerable savings can be achieved by first performing a linked-list sort and then physically rearranging the records according to the order specified in the list. This rearranging can be accomplished in linear time using some additional space.

If the list has been sorted so that at the end of the sort, *first* is a pointer to the first record in a linked list of records, then each record in this list will have a key that is greater than or equal to the key of the previous record (if there is a previous record). To physically rearrange these records into the order specified by the list, we begin by interchanging records R_1 and R_{last} . Now, the record in the position R_1 has the smallest key. If *first* $\neq 1$, then there is some record in the list whose link field is 1. If we could change this link field to indicate the new position of the record previously at position 1, then we would be left with records R_2, \dots, R_n linked together in nondecreasing order. Repeating the above process will, after $n - 1$ iterations, result in the desired rearrangement. The snag, however, is that in a singly linked list we do not know the predecessor of a node. To overcome this difficulty, our *list_rearrangement* function, *List1* (Program 7.16), begins by converting the singly linked list *first* into a doubly linked list and then proceeds to move records into their correct places. This

function assumes that links are stored in an integer array as in the case of our Radix Sort and Recursive Merge Sort functions.

```

template <class T>
void List(T *a, int *links, const int n, int first)
{ // Rearrange the sorted chain beginning at first so that the records a[1:n]
  // are in sorted order.
    int *linkb = new int [n]; // array for backward links
    int prev = 0;
    for (int current = first; current; current = links[current])
    { // convert chain into a doubly linked list
        linkb[current] = prev;
        prev = current;
    }

    for (int i = 1; i < n; i++) // move a[first] to position i while
    { // maintaining the list
        if (first != i) {
            if (links[i]) linkb[links[i]] = first;
            links[linkb[i]] = first;
            swap(a[first], a[i]);
            swap(links[first], links[i]);
            swap(linkb[first], linkb[i]);
        }
        first = links[i];
    }
}

```

Program 7.16: Rearranging records using a doubly linked list

Example 7.9: After a List Sort on the input list (26, 5, 77, 1, 61, 11, 59, 15, 48, 19) has been made, the list is linked as in Figure 7.10(a) (only the key and link fields of each record are shown). Following the links starting at *first*, we obtain the logical sequence of records $R_4, R_2, R_5, R_8, R_{18}, R_1, R_9, R_7, R_3$, and R_6 . This sequence corresponds to the key sequence 1, 5, 11, 15, 19, 26, 48, 59, 61, 33. Filling in the backward links, we get the doubly linked list of Figure 7.10(b). Figure 7.11 shows the list following the first four iterations of the second for loop of *List*. The changes made in each iteration are shown in boldface. □

Analysis of *List*: If there are n records in the list, then the time required to convert the chain *first* into a doubly linked list is $O(n)$. The second for loop is

<i>i</i>	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
<i>key</i>	26	5	77	1	61	11	39	15	48	19
<i>link</i>	9	6	0	2	3	8	5	10	7	1

(a) Linked list following a List Sort, $first = 4$

<i>i</i>	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
<i>key</i>	26	5	77	1	61	11	39	15	48	19
<i>link</i>	9	6	0	2	3	8	5	10	7	1
<i>linkb</i>	10	4	3	0	7	2	9	6	1	8

(b) Corresponding doubly linked list, $first = 4$

Figure 7.10: Sorted linked lists

iterated $n - 1$ times. In each iteration, at most two records are interchanged. This requires three record moves. If each record is m words long, then the cost per record swap is $O(m)$. The total time is therefore $O(nm)$.

The worst case of $3(n - 1)$ record moves (note that each swap requires 3 record moves) is achievable. For example, consider the input key sequence R_1, R_2, \dots, R_n , with $R_2 < R_3 < \dots < R_n$ and $R_1 > R_n$. \square

Although several modifications to *List1* are possible, one of particular interest was given by M. D. MacLaren. This modification results in a rearrangement function, *List2*, in which no additional link fields are necessary. In this function (Program 7.17), after the record $R_{j_{\text{new}}}$ is swapped with R_i , the link field of the new R_i is set to *first* to indicate that the original record was moved. This, together with the observation that *first* must always be $\geq i$, permits a correct reordering of the records.

Example 7.10: The data is the same as in Example 7.9. After the List Sort we have the configuration of Figure 7.10(a). The configuration after each of the first five iterations of the *for* loop of *List2* is shown in Figure 7.12. \square

Analysis of *list2*: The sequence of record moves for *List2* is identical to that for *list1*. Hence, in the worst case $3(n - 1)$ record moves for a total cost of $O(nm)$ are made. No node is examined more than once in the while loop. So, the total time for the while loop is $O(n)$. \square

<i>i</i>	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	1	5	77	26	61	11	59	15	48	19
linka	2	6	0	9	3	8	5	10	7	4
linkb	0	4	5	10	7	2	9	6	4	8

(a) Configuration after first iteration of the for loop of *List1*, *first* = 2

<i>i</i>	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	1	5	77	26	61	11	59	15	48	19
linka	2	6	0	9	3	8	5	10	7	4
linkb	0	4	5	10	7	2	9	6	4	8

(b) Configuration after second iteration, *first* = 6

<i>i</i>	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	1	5	11	26	61	77	59	15	48	19
linka	2	6	8	9	6	0	3	10	7	4
linkb	0	4	2	10	7	5	9	6	4	8

(c) Configuration after third iteration, *first* = 8

<i>i</i>	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	1	5	11	15	61	77	59	26	48	19
linka	2	6	8	10	6	0	5	9	7	8
linkb	0	4	2	6	7	5	9	10	8	8

(d) Configuration after fourth iteration, *first* = 10Figure 7.11: Example for *List1* (Program 7.16)

Although the asymptotic computing time for both *List1* and *List2* is the same, and the same number of record moves is made in either case, we expect *List2* to be slightly faster than *List1* because each time two records are swapped, *List1* does more work than *List2* does. *List1* is inferior to *List2* in both space and time considerations.

The List Sort technique is not well suited for Quick Sort and Heap Sort. The sequential representation of the heap is essential to Heap Sort. For these sort methods, as well as for methods suited to List Sort, one can maintain an

```

template <class T>
void List2(T *a, int *link, const int n, list first)
// Same function as List1 except that a second link array link is not required.
for (int i = 1; i <= n; i++)
// Find correct record for ith position. Its index is  $\geq i$  as
// records in positions 1, 2, ..., i - 1 are already correctly positioned.
    while (first < i) first = link[first];
    int q = link[first]; // a[q] is next record in sorted order
    if (first != i)
        // a[first] has ith smallest key. Move record to ith position.
        // Also set link from old position of a[i] to new one.
        swap(a[i], a[first]);
        link[first] = link[i];
        link[i] = first;
    }
    first = q;
}

```

Program 7.17: Rearranging records using only one link field

auxiliary table, ι , with one entry per record. The entries in this table serve as an indirect reference to the records.

At the start of the sort, $\iota[i] = i$, $1 \leq i \leq n$. If the sorting function requires a swap of $a[i]$ and $a[j]$, then only the table entries (i.e., $\iota[i]$ and $\iota[j]$) need to be swapped. At the end of the sort, the record with the smallest key is $a[\iota[1]]$ and that with the largest $a[\iota[n]]$. The required permutation on the records is $a[\iota[1]]$, $a[\iota[2]]$, ..., $a[\iota[n]]$ (see Figure 7.13). This table is adequate even in situations such as binary search, where a sequentially ordered list is needed. In other situations, it may be necessary to physically rearrange the records according to the permutation specified by ι .

The function to rearrange records corresponding to the permutation $\iota[1]$, $\iota[2]$, ..., $\iota[n]$ is a rather interesting application of a theorem from mathematics: Every permutation is made up of disjoint cycles. The cycle for any element i is made up of ι , $\iota[i]$, $\iota^2[i]$, ..., $\iota^k[i]$, where $\iota^j[i] = \iota[\iota^{j-1}[i]]$, $\iota^0[i] = i$, and $\iota^k[i] = i$. Thus, the permutation ι of Figure 7.13 has two cycles, the first involving R_3 and R_5 and the second involving R_4 , R_1 , and R_2 . Function *Table* (Program 7.18) utilizes this cyclic decomposition of a permutation. First, the cycle containing R_1 is followed, and all records are moved to their correct positions. The cycle containing R_2 is the next one examined unless this cycle has already

i	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	1	5	77	26	61	11	59	15	48	19
link	4	6	0	9	3	8	5	10	7	1

(a) Configuration after first iteration of the for loop of *List2*, *first* = 2

i	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	1	5	77	26	61	11	59	15	48	19
link	4	6	0	9	3	8	5	10	7	1

(b) Configuration after second iteration, *first* = 6

i	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	1	5	11	26	61	77	59	15	48	19
link	4	6	6	9	3	0	5	10	7	1

(c) Configuration after third iteration, *first* = 8

i	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	1	5	11	15	61	77	59	26	48	19
link	4	6	6	8	3	0	5	9	7	1

(d) Configuration after fourth iteration, *first* = 10

i	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	1	5	11	15	19	77	59	26	48	61
link	4	6	6	8	10	0	5	9	7	3

(e) Configuration after fifth iteration, *first* = 1Figure 7.12: Example for *List2* (Program 7.17)

been examined. The cycles for R_2, R_4, \dots, R_{n-1} are followed in this order. The result is a physically sorted list.

When processing a trivial cycle for R_i (i.e., $r(i) = i$), no rearrangement involving record R_i is required, since the condition $r(i) = i$ means that the record with the i th smallest key is R_i . In processing a nontrivial cycle for record R_i (i.e., $r(i) \neq i$), R_i is moved to a temporary position p , then the record at $r(i)$ is moved to i ; next the record at $r(r(i))$ is moved to $r(i)$, and so on until the end of the cycle $r^k(i)$ is reached and the record at p is moved to $r^{k+1}(i)$.

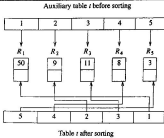


Figure 7.13: Table Sort

```

template <class T>
void Table(T *a, const int n, int *r)
// Rearrange a[1:n] to correspond to the sequence a[r[1]], ..., a[r[n]], n ≥ 1.
for (int i = 1; i < n; i++)
    if (r[i] != i) // there is a non-trivial cycle starting at i
        T p = a[r[i]];
        int j = i;
        do {
            int k = r[j]; a[j] = a[k]; r[j] = j;
            j = k;
        } while (r[j] != i);
        a[j] = p; // j is position for record p
        r[j] = j;
}

```

Program 7.18: Table Sort

Example 7.11: Suppose we start with the table t of Figure 7.14(a). This figure also shows the record keys. The table configuration is that following a Table Sort. There are two nontrivial cycles in the permutation specified by t . The first is R_1, R_3, R_4, R_4, R_1 . The second is R_4, R_5, R_7, R_4 . During the first iteration ($i = 1$) of the **for** loop of *table* (Program 7.18), the cycle $R_1, R_{t[1]} (= R_3), R_{t^2[1]} (= R_4), R_1$ is followed. Record R_1 is moved to a temporary spot p ; $R_{t[1]}$ (i.e., R_3) is moved to the position R_1 ; $R_{t^2[1]}$ (i.e., R_4) is moved to R_3 ; R_4 is moved to R_4 ; and finally p is moved to R_4 . Thus, at the end of the first iteration we have the table configuration of Figure 7.14(b).

	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8
key	35	14	12	42	26	50	31	18
t	3	2	8	5	7	1	4	6

(a) Initial configuration

key	12	14	18	42	26	35	31	50
t	1	2	3	5	7	6	4	8

(b) Configuration after rearrangement of first cycle

key	12	14	18	26	31	35	42	50
t	1	2	3	4	5	6	7	8

(c) Configuration after rearrangement of second cycle

Figure 7.14: Table Sort example

For $i = 2$ or 3 , $t[i] = i$, indicating that these records are already in their correct positions. When $i = 4$, the next nontrivial cycle is discovered, and the records on this cycle (R_4, R_5, R_7, R_4) are moved to their correct positions. Following this we have the table configuration of Figure 7.14(c).

For the remaining values of i ($i = 5, 6$, and 7), $t[i] = i$, and no more nontrivial cycles are found. \square

Analysis of Table: If each record uses m words of storage, then the additional space needed is m words for p plus a few more for variables such as i, j , and k . To obtain an estimate of the computing time, we observe that the **for** loop is executed $n - 1$ times. If for some value of i , $t[i] \neq i$, then there is a nontrivial cycle including $k > 1$ distinct records $R_i, R_{t[i]}, \dots, R_{t^{k-1}[i]}$. Rearranging these records requires $k + 1$ record moves. Following this, the records involved in this cycle

are not moved again at any time in the algorithm, as $r[j] = j$ for all such records R_j . Hence, no record can be in two different nontrivial cycles. Let k_i be the number of records on a nontrivial cycle starting at R_i when $i = i$ in the for loop. Let $k_i = 0$ for a trivial cycle. The total number of record moves is

$$\sum_{i=0, k_i \neq 0}^{n-1} (k_i + 1)$$

Since the records on nontrivial cycles must be different, $\sum k_i \leq n$. The total number of record moves is maximum when $\sum k_i = n$ and there are $\lfloor n/2 \rfloor$ cycles. When n is even, each cycle contains two records. Otherwise, one cycle contains three and the others two each. In either case the number of record moves is $\lfloor 3n/2 \rfloor$. One record move costs $O(m)$ time. The total computing time is therefore $O(mn)$. \square

Comparing *List2* (Program 7.17) and *Table*, we see that in the worst case, *List2* makes $3(n-1)$ record moves, whereas *Table* makes only $\lfloor 3n/2 \rfloor$ record moves. For larger values of n it is worthwhile to make one pass over the sorted list of records, creating a table r corresponding to a *Table Sort*. This would take $O(n)$ time. Then *Table* could be used to rearrange the records in the order specified by r .

EXERCISES

1. Complete Example 7.9.
2. Complete Example 7.10.
3. Write a version of Selection Sort (see Chapter 1) that works on a chain of records.
4. Write a *Table Sort* version of Quick Sort. Now during the sort, records are not physically moved. Instead, $r[i]$ is the index of the record that would have been in position i if records were physically moved around as in *QuickSort* (Program 7.6). Begin with $r[i] = i$, $1 \leq i \leq n$. At the end of the sort, $r[i]$ is the index of the record that should be in the i th position in the sorted list. So now function *Table* may be used to rearrange the records into the sorted order specified by r . Note that this reduces the amount of data movement taking place when compared to *QuickSort* for the case of large records.
5. Do Exercise 4 for the case of Insertion Sort.
6. Do Exercise 4 for the case of Merge Sort.
7. Do Exercise 4 for the case of Heap Sort.

7.9 SUMMARY OF INTERNAL SORTING

Of the several sorting methods we have studied, no one method is best under all circumstances. Some methods are good for small n , others for large n . Insertion Sort is good when the list is already partially ordered. Because of the low overhead of the method, it is also the best sorting method for "small" n . Merge Sort has the best worst-case behavior but requires more storage than Heap Sort. Quick Sort has the best average behavior, but its worst-case behavior is $O(n^2)$. The behavior of Radix Sort depends on the size of the keys and the choice of r . Figure 7.15 summarizes the asymptotic complexity of the first four of these sort methods.

Method	Worst	Average
Insertion Sort	n^3	n^2
Heap Sort	$n \log n$	$n \log n$
Merge Sort	$n \log n$	$n \log n$
Quick Sort	n^2	$n \log n$

Figure 7.15: Comparison of sort methods

Figure 7.16 gives the average runtimes for the four sort methods of Figure 7.15. These times were obtained on a 1.7GHz Intel Pentium 4 PC with 512 MB RAM and Microsoft Visual Studio .NET 2003. For each n at least 100 randomly generated integer instances were run. These random instances were constructed by making repeated calls to the C++ function `rand`. If the time taken to sort these instances was less than 1 second then additional random instances were sorted until the total time taken was at least this much. The times reported in Figure 7.16 include the time taken to set up the random data. For each n the time taken to set up the data and the time for the remaining overheads included in the reported numbers is the same for all sort methods. As a result, the data of Figure 7.16 is useful for comparative purposes. The data of this figure is plotted in Figure 7.17.

As Figure 7.17 shows, Quick Sort outperforms the other sort methods for visibly large n . We see that the break-even point between Insertion and Quick Sort is between 50 and 100. The exact break-even point can be found experimentally by obtaining run-time data for n between 50 and 100. Let the exact break-even point be n_{break} . For average performance, Insertion Sort is the best sort method (of those tested) to use when $n < n_{\text{break}}$, and Quick Sort is the best

<i>n</i>	<i>Insert</i>	<i>Heap</i>	<i>Merge</i>	<i>Quick</i>
0	0.000	0.000	0.000	0.000
50	0.004	0.009	0.008	0.006
100	0.011	0.019	0.017	0.013
200	0.033	0.042	0.037	0.029
300	0.067	0.066	0.059	0.045
400	0.117	0.090	0.079	0.061
500	0.179	0.116	0.100	0.079
1000	0.662	0.245	0.213	0.169
2000	2.439	0.519	0.459	0.358
3000	5.390	0.809	0.721	0.560
4000	9.530	1.105	0.972	0.761
5000	15.935	1.410	1.271	0.970

Times are in milliseconds

Figure 7.16: Average times for sort methods

when $n > n_{\text{break}}$. We can improve on the performance of Quick Sort for $n > n_{\text{break}}$ by combining Insertion and Quick Sort into a single sort function by replacing the following statement in Program 7.19

```
if (left < right) {
    code to partition and make recursive calls
}
```

with the code

```
if (left + nBreak < right) {
    code to partition and make recursive calls
}
else {
    sort  $s[left:right]$  using Insertion Sort;
    return;
}
```

For worst-case behavior most implementations will show Merge Sort to be best for $n > c$ where c is some constant. For $n \leq c$ Insertion Sort has the best worst-case behavior. The performance of Merge Sort can be improved by combining Insertion Sort and Merge Sort in a manner similar to that described above for combining Insertion Sort and Quick Sort.

The run-time results for the sort methods point out some of the limitations

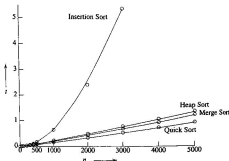


Figure T.17: Plot of average times (milliseconds)

of asymptotic complexity analysis. Asymptotic analysis is not a good predictor of performance for small instances—insertion sort with its $O(n^2)$ complexity is better than all of the $O(n \log n)$ methods for small instances. Programs that have the same asymptotic complexity often have different actual runtimes.

C++'s Sort Methods

If you had to develop the STL sort function `sort`, what would you do? Should you pick a method that optimizes worst-case performance, or should you pick one that optimizes average performance? Should you limit your choice to a sort method that is stable (i.e., a method that does not alter the relative order of equal elements)?

The developers of C++'s sort function opted to optimize average performance and use a modified Quick Sort that reverts to a Heap Sort when the number of subdivisions exceeds some constant times $\log_2 n$ and to an Insertion Sort when the segment size becomes small. The STL function `stable_sort` is a

Merge Sort that reverts to an Insertion Sort when the segment size becomes small. The STL function `partial_sort` is based on Heap Sort and has the ability to stop when only the first k elements need to be sorted.

EXERCISES

1. [Cower Sort] The simplest known sorting method arises from the observation that the position of a record in a sorted list depends on the number of records with smaller keys. Associated with each record there is a *count* field used to determine the number of records that must precede this one in the sorted list. Write a function to determine the count of each record in an unordered list. Show that if the list has n records, then all the counts can be determined by making at most $n(n-1)/2$ key comparisons.
2. Write a function similar to `Table` (Program 7.18) to rearrange the records of a list if, with each record, we have a count of the number of records preceding it in the sorted list (see Exercise 1).
3. Obtain Figures 7.18 and 7.19 for the worst-case runtime.
4. [Programming Project] The objective of this assignment is to come up with a composite sorting function that is good on the worst-time criterion. The candidate sort methods are (a) Insertion Sort, (b) Quick Sort, (c) Merge Sort, (d) Heap Sort.

To begin with, program these sort methods in C++. In each case, assume that n integers are to be sorted. In the case of Quick Sort, use the median-of-three method. In the case of Merge Sort, use the iterative method (as a separate exercise, you might wish to compare the runtimes of the iterative and recursive versions of Merge Sort and determine what the recursion penalty is in your favorite language using your favorite compiler). Check out the correctness of the programs using some test data. Since quite detailed and working functions are given in the book, this part of the assignment should take little effort. In any case, no points are earned until after this step.

To obtain reasonably accurate runtimes, you need to know the accuracy of the clock or timer you are using. Determine this by reading the appropriate manual. Let the clock accuracy be δ . Now, run a pilot test to determine ballpark times for your four sorting functions for $n = 500, 1000, 2000, 3000, 4000$, and 5000 . You will notice times of 0 for many of these values of n . The other times may not be much larger than the clock accuracy.

To time an event that is smaller than or near the clock accuracy, repeat it many times and divide the overall time by the number of repetitions. You should obtain times that are accurate to within 1%.

We need worst-case data for each of the four sort methods. The worst-case data for Insertion Sort are easy to generate. Just use the sequence $n, n-1, n-2, \dots, 1$. Worst-case data for Merge Sort can be obtained by working backward. Begin with the last merge your function will perform and make this work hardest. Then look at the second-to-last merge, and so on. Use this logic to obtain a program that will generate worst-case data for Merge Sort for each of the above values of n .

Generating worst-case data for Heap Sort is the hardest, so, here we shall use a random permutation generator (one is provided in Program 7.20). We shall generate random permutations of the desired size, clock Heap Sort on each of these, and use the max of these times to approximate to the worst-case time. You will be able to use more random permutations for smaller values of n than for larger. For no value of n should you use fewer than 10 permutations. Use the same technique to obtain worst-case times for Quick Sort.

```
template <class T>
void Permute(T *a, int n)
// Random permutation generator.
for (int i = n; i >= 2; i--)
{
    int j = rand() % (i + 1); // j = random integer in the range [1, i]
    swap(a[j], a[i]);
}
}
```

Program 7.20: Random permutation generator

Having sorted on the test data, we are ready to perform our experiment. Obtain the worst-case times. From these times you will get a rough idea when one function performs better than the other. Now, narrow the scope of your experiments and determine the exact value of n when one sort method outperforms another. For some methods, this value may be 0. For instance, each of the other three methods may be faster than Quick Sort for all values of n .

Plot your findings on a single sheet of graph paper. Do you see the n^2 behavior of Insertion Sort and Quick Sort and the $n \log n$ behavior of the other two methods for suitably large n (about $n > 20$)? If not, there is something wrong with your test or your clock or with both. For each value of n determine the sort function that is fastest (simply look at your graph). Write a composite function with the best possible performance for all n .

Clock this function and plot the times on the same graph sheet you used earlier.

WHAT TO TURN IN

You are required to submit a report that states the clock accuracy, the number of random permutations tried for Heap Sort, the worst-case data for Merge Sort and how you generated it, a table of times for the above values of n , the times for the narrowed ranges, the graph, and a table of times for the composite function. In addition, your report must be accompanied by a complete listing of the program used by you (this includes the sorting functions and the main program for timing and test-data generation).

5. Repeat the previous exercise for the case of average runtimes. Average-case data are usually very difficult to create, so use random permutations. This time, however, do not repeat a permutation many times to overcome clock inaccuracies. Instead, use each permutation once and clock the overall time (for a fixed n).
6. Assume you are given a list of five-letter English words and are faced with the problem of listing these words in sequences such that the words in each sequence are anagrams (i.e., if x and y are in the same sequence, then word x is a permutation of word y). You are required to list out the fewest such sequences. With this restriction, show that no word can appear in more than one sequence. How would you go about solving this problem?
7. Assume you are working in the census department of a small town where the number of records, about 3000, is small enough to fit into the internal memory of a computer. All the people currently living in this town were born in the United States. There is one record for each person in this town. Each record contains (a) the state in which the person was born, (b) county of birth, and (c) name of person. How would you produce a list of all persons living in this town? The list is to be ordered by state. Within each state the persons are to be listed by their counties, the counties being arranged in alphabetical order. Within each county, the names are also listed in alphabetical order. Justify any assumptions you make.
8. [Bubble Sort] In a Bubble Sort several left-to-right passes are made over the array of records to be sorted. In each pass, pairs of adjacent records are compared and exchanged if necessary. The sort terminates following a pass in which no records are exchanged.
 - (a) Write a C++ function for Bubble Sort.
 - (b) What is the worst-case complexity of your function?
 - (c) How much time does your function take on a sorted array of records?
 - (d) How much time does your function take on an array of records that are in the reverse of sorted order?

438 Sorting

9. Redo the preceding exercise beginning with an unsorted chain of records and ending with a sorted chain.
10. [Programming Project] The objective of this exercise is to study the effect of the size of an array element on the computational time of various sorting algorithms.
 - (a) Use Insertion Sort, Quick Sort, Iterative Merge Sort, and Heap Sort to sort arrays of (i) characters (`char`), (ii) integers (`int`), (iii) floating point numbers (`float`), and (iv) rectangles (Assume that a rectangle is represented by the coordinates of its bottom left point and its height and width, all of which are of type `float`. Assume, also, that rectangles are to be sorted in non-decreasing order of their areas.)
 - (b) Obtain a set of runtimes for each algorithm-data type pair specified above. (There should be sixteen such pairs.) To obtain a set of runtimes of an algorithm-data type pair, you should run the algorithm on at least four arrays of different sizes containing elements of the appropriate data type. The elements in an array should be generated using a random number generator.
 - (c) Draw tables and graphs displaying your experimental results. What do you conclude from the experiments?

7.10 EXTERNAL SORTING

7.10.1 Introduction

In this section, we assume that the lists to be sorted are so large that the whole list cannot be contained in the internal memory of a computer, making an internal sort impossible. We shall assume that the list (or file) to be sorted resides on a disk. The term *block* refers to the unit of data that is read from or written to a disk at one time. A block generally consists of several records. For a disk, there are three factors contributing to the read/write time:

- (1) *Seek time*: time taken to position the read/write heads to the correct cylinder. This will depend on the number of cylinders across which the heads have to move.
- (2) *Latency time*: time until the right sector of the track is under the read/write head.
- (3) *Transmission time*: time to transmit the block of data to/from the disk.

The most popular method for sorting on external storage devices is Merge Sort. This method consists of two distinct phases. First, segments of the input

list are sorted using a good internal sort method. These sorted segments, known as runs, are written onto external storage as they are generated. Second, the runs generated in phase one are merged together following the merge-tree pattern of Figure 7.4, until only one run is left. Because the simple merge function *merge* (Program 7.7) requires only the leading records of the two runs being merged to be present in memory at one time, it is possible to merge large runs together. It is more difficult to adapt the other internal sort methods considered in this chapter to external sorting.

Example 7.12: A list containing 4500 records is to be sorted using a computer with an internal memory capable of sorting at most 750 records. The input list is maintained on disk and has a block length of 250 records. We have available another disk that may be used as a scratch pad. The input disk is not to be written on. One way to accomplish the sort using the general function outlined above is to

(1) Internally sort three blocks at a time (i.e., 750 records) to obtain six runs R_1 to R_6 . A method such as Heap Sort, Merge Sort, or Quick Sort could be used. These six runs are written onto the scratch disk (Figure 7.20).

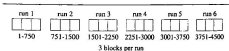


Figure 7.20: Blocked runs obtained after internal sorting

(2) Set aside three blocks of internal memory, each capable of holding 250 records. Two of these blocks will be used as input buffers and the third as an output buffer. Merge runs R_1 and R_2 . This merge is carried out by first reading one block of each of these runs into input buffers. Blocks of runs are merged from the input buffers into the output buffer. When the output buffer gets full, it is written onto the disk. If an input buffer gets empty, it is refilled with another block from the same run. After runs R_1 and R_2 are merged, R_3 and R_4 are merged, and finally R_5 and R_6 are merged. The result of this pass is three runs, each containing 1500 sorted records or six blocks. Two of these runs are now merged using the input/output buffers set up as above to obtain a run of size 3000. Finally, this run is merged with the remaining run of size 1500 to obtain the desired sorted list (Figure 7.21). □

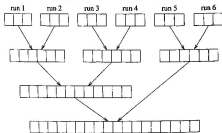


Figure 7.21: Merging the six runs

To analyze the complexity of external sort, we use the following notation:

t_s = maximum seek time

t_l = maximum latency time

t_{rw} = time to read or write one block of 250 records

t_{io} = time to input or output one block

$$= t_s + t_l + t_{rw}$$

t_{is} = time to internally sort 750 records

m_m = time to merge n records from input buffers to the output buffer

We shall assume that each time a block is read from or written onto the disk, the maximum seek and latency times are experienced. Although this is not true in general, it will simplify the analysis. The computing times for the various operations in our 4500-record example are given in Figure 7.22.

The contribution of seek time can be reduced by writing blocks on the same cylinder or on adjacent cylinders. A close look at the final computing time indicates that it depends chiefly on the number of passes made over the data. In addition to the initial input pass made over the data for the internal sort, the

operation	time
(1) read 18 blocks of input, $18t_{IO}$, internally sort, $6t_{IS}$, write 18 blocks, $18t_{IO}$	$36t_{IO} + 6t_{IS}$
(2) merge runs 1 to 6 in pairs	$36t_{IO} + 4500t_m$
(3) merge two runs of 1500 records each, 12 blocks	$24t_{IO} + 3000t_m$
(4) merge one run of 3000 records with one run of 1500 records	$36t_{IO} + 4500t_m$
total time	$132t_{IO} + 12,000t_m + 6t_{IS}$

Figure 7.22: Computing times for disk sort example

merging of the runs requires 2-2/3 passes over the data (one pass to merge 6 runs of length 750 records, two-thirds of a pass to merge two runs of length 1500, and one pass to merge one run of length 3000 and one of length 1500). Since one full pass covers 18 blocks, the input and output time is $2 \times (2\text{-}2/3 + 1) \times 18 t_{IO} = 132t_{IO}$. The leading factor of 2 appears because each record that is read is also written out again. The merge time is $2\text{-}2/3 \times 4500t_m = 12,000t_m$. Because of this close relationship between the overall computing time and the number of passes made over the data, future analysis will be concerned mainly with counting the number of passes being made. Another point to note regarding the above sort is that no attempt was made to use the computer's ability to carry out input/output and CPU operation in parallel and thus overlap some of the time. In the ideal situation we would overlap almost all the input/output time with CPU processing so that the real time would be approximately $132 t_{IO} = 12,000 t_m + 6t_{IS}$.

If we have two disks, we can write on one, read from the other, and merge buffer loads already in memory in parallel. A proper choice of buffer lengths and buffer handling schemes will result in a time of almost $66t_{IO}$. This parallelism is an important consideration when sorting is being carried out in a nonmultiprogramming environment. In this situation, unless input/output and CPU processing is going on in parallel, the CPU is idle during input/output. In a multiprogramming environment, however, the need for the sorting program to carry out input/output and CPU processing in parallel may not be so critical, since the CPU can be busy working on another program (if there are other programs in the

system at the time) while the sort program waits for the completion of its input/output. Indeed, in many multiprogramming environments it may not even be possible to achieve parallel input, output, and internal computing because of the structure of the operating system.

The number of merge passes over the runs can be reduced by using a higher-order merge than two-way merge. To provide for parallel input, output, and merging, we need an appropriate buffer-handling scheme. Further improvement in runtime can be obtained by generating fewer (or equivalently longer) runs than are generated by the strategy described above. This can be done using a loser tree. The loser-tree strategy to be discussed in Section 7.10.4 results in runs that are on the average almost twice as long as those obtained by the above strategy. However, the generated runs are of varying size. As a result, the order in which the runs are merged affects the time required to merge all runs into one. We consider these factors now.

7.10.2 *k*-Way Merging

The two-way merge function *Merge* (Program 7.7) is almost identical to the merge function just described (Figure 7.21). In general, if we start with m runs, the merge tree corresponding to Figure 7.21 will have $\lceil \log_2 m \rceil + 1$ levels, for a total of $\lceil \log_2 m \rceil$ passes over the data list. The number of passes over the data can be reduced by using a higher-order merge (i.e., k -way merge for $k \geq 2$). In this case, we would simultaneously merge k runs together. Figure 7.23 illustrates a four-way merge of 16 runs. The number of passes over the data is now two, versus four passes in the case of a two-way merge. In general, a k -way merge on m runs requires $\lceil \log_k m \rceil$ passes over the data. Thus, the input/output time may be reduced by using a higher-order merge.

The use of a higher-order merge, however, has some other effects on the sort. To begin with, k runs of size $s_1, s_2, s_3, \dots, s_k$ can no longer be merged internally in $O(\sum s_i)$ time. In a k -way merge, as in a two-way merge, the next record to be output is the one with the smallest key. The smallest has now to be found from k possibilities and it could be the leading record in any of the k runs. The most direct way to merge k runs is to make $k - 1$ comparisons to determine the next record to output. The computing time for this is $O((k - 1) \sum s_i)$. Since

$\log_k m$ passes are being made, the total number of key comparisons is $n(k - 1)\log_k m = n(k - 1)\log_2 m / \log_2 k$, where n is the number of records in the list. Hence, $(k - 1)/\log_2 k$ is the factor by which the number of key comparisons increases. As k increases, the reduction in input/output time will be outweighed by the resulting increase in CPU time needed to perform the k -way merge.

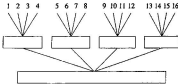


Figure 7.23: A four-way merge on 16 runs

For large k (say, $k \geq 6$) we can achieve a significant reduction in the number of comparisons needed to find the next smallest element by using a loser tree with k leaves (see Chapter 5). In this case, the total time needed per level of the merge tree is $O(n \log_2 k)$. Since the number of levels in this tree is $O(\log_2 n)$, the asymptotic internal processing time becomes $O(n \log_2 k \log_2 n) = O(n \log_2 n)$. This is independent of k .

In going to a higher-order merge, we save on the amount of input/output being carried out. There is no significant loss in internal processing speed. Even though the internal processing time is relatively insensitive to the order of the merge, the decrease in input/output time is not as much as indicated by the reduction to $\log_2 n$ passes. This is so because the number of input buffers needed to carry out a k -way merge increases with k . Although $k + 1$ buffers are sufficient, in the next section we shall see that the use of $2k + 2$ buffers is more desirable. Since the internal memory available is fixed and independent of k , the buffer size must be reduced as k increases. This in turn implies a reduction in the block size on disk. With the reduced block size, each pass over the data results in a greater number of blocks being written or read. This represents a potential increase in input/output time from the increased contribution of seek and latency times involved in reading a block of data. Hence, beyond a certain k value the input/output time will increase despite the decrease in the number of passes being made. The optimal value for k depends on disk parameters and the amount of internal memory available for buffers.

7.10.3 Buffer Handling for Parallel Operation

If k runs are being merged together by a k -way merge, then we clearly need at least k input buffers and one output buffer to carry out the merge. This, however, is not enough if input, output, and internal merging are to be carried out in parallel. For instance, while the output buffer is being written out, internal merging has to be halted, since there is no place to collect the merged records. This can be overcome through the use of two output buffers. While one is being written out, records are merged into the second. If buffer sizes are chosen correctly, then the time to output one buffer will be the same as the CPU time needed to fill the second buffer. With only k input buffers, internal merging will have to be held up whenever one of these input buffers becomes empty and another block from the corresponding run is being read in. This input delay can also be avoided if we have $2k$ input buffers. These $2k$ input buffers have to be used cleverly to avoid reaching a situation in which processing has to be held up because of a lack of input records from any one run. Simply assigning two buffers per run does not solve the problem.

Example 7.13: Assume that a two-way merge is carried out using four input buffers, $in[i]$, $0 \leq i \leq 3$, and two output buffers, $ou[0]$ and $ou[1]$. Each buffer is capable of holding two records. The first few records of run 0 have key value 1, 3, 5, 7, 8, 9. The first few records of run 1 have key value 2, 4, 6, 15, 20, 25. Buffers $in[0]$ and $in[2]$ are assigned to run 0. The remaining two input buffers are assigned to run 1. We start the merge by reading in one buffer load from each of the two runs. At this time the buffers have the configuration of Figure 7.24(a). Now runs 0 and 1 are merged using records from $in[0]$ and $in[1]$. In parallel with this, the next buffer load from run 0 is input. If we assume that buffer lengths have been chosen such that the times to input, output, and generate an output buffer are all the same, then when $ou[0]$ is full, we have the situation of Figure 7.24(b). Next, we simultaneously output $ou[0]$, input into $in[3]$ from run 1, and merge into $ou[1]$. When $ou[1]$ is full, we have the situation of Figure 7.24(c). Continuing in this way, we reach the configuration of Figure 7.24(e). We now begin to output $ou[1]$, input from run 0 into $in[2]$, and merge into $ou[0]$. During the merge, all records from run 0 get used before $ou[0]$ gets full. Merging must now be delayed until the inputting of another buffer load from run 0 is completed. \square

Example 7.13 makes it clear that if $2k$ input buffers are so suffice, then we cannot assign two buffers per run. Instead, the buffer must be floating in the sense that an individual buffer may be assigned to any run depending upon need. In the buffer assignment strategy we shall describe, there will at any time be at least one input buffer containing records from each run. The remaining buffers will be filled on a priority basis (i.e., the run for which the k -way merging algorithm will

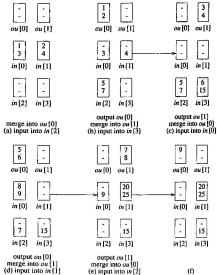


Figure 7.24: Example showing that two fixed buffers per run are not enough for continued parallel operation

run out of records first is the one from which the next buffer will be filled). One may easily predict which run's records will be exhausted first by simply comparing the keys of the last record read from each of the k runs. The smallest such key determines this run. We shall assume that in the case of equal keys, the merge process first merges the record from the run with least index. This means that if the key of the last record read from run i is equal to the key of the last record read from run j , and $i < j$, then the records read from i will be exhausted before those from j . So, it is possible to have more than two bufferloads from a given run and only one partially full buffer from another run. All bufferloads from the same run are queued together. Before formally presenting the algorithm for buffer utilization, we make the following assumptions about the parallel processing capabilities of the computer system available:

- (1) We have two disk drives and the input/output channel is such that we can simultaneously read from one disk and write onto the other.
- (2) While data transmission is taking place between an input/output device and a block of memory, the CPU cannot make references to that same block of memory. Thus, it is not possible to start filling the front of an output buffer while it is being written out. If this were possible, then by coordinating the transmission and merging rate, only one output buffer would be needed. By the time the first record for the new output block is determined, the first record of the previous output block has been written out.
- (3) To simplify the discussion we assume that input and output buffers are of the same size.

Keeping these assumptions in mind, we provide a high-level description of the algorithm obtained using the strategy outlined earlier and then illustrate how it works through an example. Our algorithm, Program 7.21, merges k -runs, $k \geq 2$, using a k -way merge. $2k$ input buffers and two output buffers are used. Each buffer is a continuous block of memory. Input buffers are queued in k queues, one queue for each run. It is assumed that each input/output buffer is long enough to hold one block of records. Empty buffers are placed on a linked stack. The algorithm also assumes that the end of each run has a sentinel record with a very large key, say $+\infty$. It is assumed that all other records have key value less than that of the sentinel record. If block lengths, and hence buffer lengths, are chosen such that the time to merge one output buffer load equals the time to read a block, then almost all input, output, and computation will be carried out in parallel. It is also assumed that in the case of equal keys, the k -way merge algorithm first outputs the record from the run with the smallest index.

We make the following observations about Program 7.21:

- (1) For large k , determination of the queue that will be exhausted first can be

(Steps in buffering algorithm)

- Step 1:** Input the first block of each of the k runs, setting up k linked queues, each having one block of data. Put the remaining k input blocks into a linked stack of free input blocks. Set ow to 0.
- Step 2:** Let $lastKey[i]$ be the last key input from run i . Let $nextRun$ be the run for which $lastKey$ is minimum. If $lastKey[nextRun] \neq +\infty$, then initiate the input of the next block from run $nextRun$.
- Step 3:** Use a function *Kwaymerge* to merge records from the k input queues into the output buffer ow . Merging continues until either the output buffer gets full or a record with key $+\infty$ is merged into ow . If, during this merge, an input buffer becomes empty before the output buffer gets full or before $+\infty$ is merged into ow , the *Kwaymerge* advances to the next buffer on the same queue and returns the empty buffer to the stack of empty buffers. However, if an input buffer becomes empty at the same time as the output buffer gets full or $+\infty$ is merged into ow , the empty buffer is left on the queue, and *Kwaymerge* does not advance to the next buffer on the queue. Rather, the merge terminates.
- Step 4:** Wait for any ongoing disk input/output to complete.
- Step 5:** If an input buffer has been read, add it to the queue for the appropriate run. Determine the next run to read from by determining $NextRun$ such that $lastKey[nextRun]$ is minimum.
- Step 6:** If $lastKey[nextRun] \neq +\infty$, then initiate reading the next block from run $nextRun$ into a free input buffer.
- Step 7:** Initiate the writing of output buffer ow . Set ow to $1 - ow$.
- Step 8:** If a record with key $+\infty$ has been not been merged into the output buffer, go back to Step 3. Otherwise, wait for the ongoing write to complete and then terminate.
-

Program 7.21: k -way merge with floating buffers

found in $\log_2 k$ comparisons by setting up a loser tree for $last[i]$, $0 \leq i < k$, rather than making $k - 1$ comparisons each time a buffer load is to be read in. The change in computing time will not be significant, since this queue selection represents only a very small fraction of the total time taken by the algorithm.

- (2) For large k , function *Kwaymerge* uses a tree of losers (see Chapter 5).
- (3) All input and output except for the input of the initial k blocks and the

output of the last block is done concurrently with computing. Since, after k runs have been merged, we would probably begin to merge another set of k runs, the input for the next set can commence during the final merge stages of the present set of runs. That is, when $\text{lastKey}[\text{nextRun}] = +\infty$ in Step 6, we begin reading one by one the first blocks from each of the next set of k runs to be merged. So, over the entire sorting of a file the only time that is not overlapped with the internal merging time is the time to input the first k blocks and that to output the last block.

- (4) The algorithm assumes that all blocks are of the same length. Ensuring this may require inserting a few dummy records into the last block of each run following the sentinel record with key $+\infty$.

Example 7.14: To illustrate the algorithm of Program 7.21, let us trace through it while it performs a three-way merge on the three runs of Figure 7.25. Each run consists of four blocks of two records each; the last key in the fourth block of each of these three runs is $+\infty$. We have six input buffers and two output buffers. Figure 7.26 shows the status of the input buffer queues, the run from which the next block is being read, and the output buffer being output at the beginning of each iteration of the loop of Steps 3 through 8 of the buffering algorithm.

Run 0	<div>20 25</div>	<div>26 28</div>	<div>29 30</div>	<div>33 $+\infty$</div>
Run 1	<div>23 29</div>	<div>34 36</div>	<div>38 60</div>	<div>70 $+\infty$</div>
Run 2	<div>24 28</div>	<div>31 33</div>	<div>40 43</div>	<div>50 $+\infty$</div>

Figure 7.25: Three runs

From line 5 of Figure 7.26 it is evident that during the k -way merge, the test for "output buffer full?" should be carried out before the test "input buffer empty?", as the next input buffer for that run may not have been read in yet, so there would be no next buffer in that queue. In lines 3 and 4 all six input buffers are in use, and the stack of free buffers is empty. \square

We end our discussion of buffer handling by proving that Program 7.21 is correct.

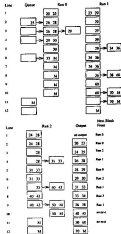


Figure 7.26: Buffering example

Theorem 7.2: The following are true for Program 7.21:

- (1) In Step 6, there is always a buffer available in which to begin reading the next block.
- (2) During the k -way merge of Step 3, the next block in the queue has been read in by the time it is needed.

Proof: (1) Each time we get to Step 6 of the algorithm, there are at most $k + 1$ buffer loads in memory, one of these being in an output buffer. For each queue there can be at most one buffer that is partially full. If no buffer is available for the next read, then the remaining k buffers must be full. This means that all the k partially full buffers are empty (as otherwise there will be more than $k + 1$ buffer loads in memory). From the way the merge is set up, only one buffer can be both unavailable and empty. This may happen only if the output buffer gets full exactly when one input buffer becomes empty. But $k > 1$ contradicts this. So, there is always at least one buffer available when Step 6 is being executed.

(2) Assume this is false. Let run R_i be the one whose queue becomes empty during *Kwaymerge*. We may assume that the last key merged was not $+\infty$, since otherwise *Kwaymerge* would terminate the merge rather than get another buffer for R_i . This means that there are more blocks of records for run R_i on the input file, and $\text{lastKey}[i] \neq +\infty$. Consequently, up to this time whenever a block was output, another was simultaneously read in. Input and output therefore proceeded at the same rate, and the number of available blocks of data was always k . An additional block is being read in, but it does not get queued until Step 5. Since the queue for R_i has become empty first, the selection rule for choosing the next run to read from ensures that there is at most one block of records for each of the remaining $k - 1$ runs. Furthermore, the output buffer cannot be full at this time, as this condition is tested for before the input-buffer-empty condition. Thus, fewer than k blocks of data are in memory. This contradicts our earlier assertion that there must be exactly k such blocks of data. \square

7.10.4 Run Generation

Using conventional internal sorting methods such as those discussed earlier in this chapter, it is possible to generate runs that are only as large as the number of records that can be held in internal memory at one time. Using a tree of losers, it is possible to do better than this. In fact, the algorithm we shall present will, on the average, generate runs that are twice as long as obtainable by conventional methods. This algorithm was devised by Walters, Painter, and Zalka. In addition to being capable of generating longer runs, this algorithm will allow for parallel input, output, and internal processing.

We assume that input/output buffers have been set up appropriately for maximum overlapping of input, output, and internal processing. Whenever there is an input/output instruction in the run-generation algorithm, it is assumed that the operation takes place through the input/output buffers. The run generation algorithm uses a tree of losers. We assume that there is enough space to construct such a tree for k records, $r[i]$, $0 \leq i < k$. Each node, i , in this tree has one field $l[i]$. $l[i] \in \{1 \leq i < k\}$, represents the loser of the tournament played at node i .

Each of the k record positions $r[i]$ has a run number $rn[i]$, $0 \leq i < k$. This field enables us to determine whether or not $r[i]$ can be output as part of the run currently being generated. Whenever the tournament winner is output, a new record (if there is one) is input, and the tournament is replayed as discussed in Chapter 5.

Function *Runs* (Program 7.22) is an implementation of the loser tree strategy just discussed. The variables used in this function have the following significance:

$r[i]$, $0 \leq i < k$...	the k records in the tournament tree
$l[i]$, $1 \leq i < k$...	loser of the tournament played at node i
$l[0]$...	winner of the tournament
$rn[i]$, $0 \leq i < k$...	the run number to which $r[i]$ belongs
rc	...	run number of current run
q	...	overall tournament winner
rq	...	run number for $r[q]$
$rmux$...	number of runs that will be generated
$lastRec$...	last record output
$MAXREC$...	a record with maximum key possible

The function assumes that the relational operators have been overloaded so that when two records of the template type T are compared, the comparison is done using their keys.

The loop of lines 11 to 34 repeatedly plays the tournament outputting records. The variable *lastKey* is made use of in line 22 to determine whether or not the new record input, $r[q]$, can be output as part of the current run. If $key[q] < lastKey$ then $r[q]$ cannot be output as part of the current run rc , as a record with larger key value has already been output in this run. When the tree is being readjusted (lines 27 to 33), a record with lower run number wins over one with a higher run number. When run numbers are equal, the record with lower key value wins. This ensures that records come out of the tree in nondecreasing order of their run numbers. Within the same run, records come out of the tree in nondecreasing order of their key values. *rmux* is used to terminate the function. In line 19, when we run out of input, a record with run number $rmux + 1$ is introduced. When this record is ready for output, the function terminates from line 14.

Analysis of Runs: When the input list is already sorted, only one run is generated. On the average, the run size is almost $2k$. The time required to generate all the runs for an n run list is $O(n \log k)$, as it takes $O(\log k)$ time to adjust the loser tree each time a record is output. \square

```

template <class T>
1 void Run(T *r)
2 {
3     r = new T[k];
4     int *rn = new int[k], *l = new int[k];
5     for (int i = 0; i < k; i++) { // input records
6         InputRecord(r[i]); rn[i] = 1;
7     }
8     InitializeLoserTree();
9     T q = l[0]; // tournament winner
10    int rq = 1, rc = 1, rmax = 1; T lastRec = MAXREC;
11    while(1) { // output runs
12        if (rq != rc) { // end of run
13            output end of run marker;
14            if (rq > rmax) return;
15            else rc = rq;
16        }
17        WriteRecord(r[q]); lastRec = r[q]; // output record r[q]
18        // input new record into tree
19        if (end of input) rn[q] = rmax + 1;
20        else {
21            ReadRecord(r[q]);
22            if (r[q] < lastRec) // new record belongs to next run
23                rn[q] = rmax = rq + 1;
24            else rn[q] = rc;
25        }
26        rq = rn[q];
27        // adjust losers
28        for (t = (k+q)/2; t; t /= 2) // t is initialized to be parent of q
29            if ((rn[t/r] < rq) || ((rn[t/r] == rq) && (r[t/r] < r[q])))
30                // t is the winner
31                swap(q, t/r);
32        rq = rn[q];
33    }
34 }
35 delete [] r; delete [] rn; delete [] l;
36 }

```

Program 7.22: Run generation using a loser tree

7.10.5 Optimal Merging of Runs

The runs generated by function *Riser* may not be of the same size. When runs are of different size, the run merging strategy employed so far (i.e., make complete passes over the collection of runs) does not yield minimum runtimes. For example, suppose we have four runs of length 2, 4, 5, and 15, respectively. Figure 7.27 shows two ways to merge these using a series of two-way merges. The circular nodes represent a two-way merge using as input the data of the children nodes. The square nodes represent the initial runs. We shall refer to the circular nodes as *internal nodes* and the square ones as *external nodes*. Each figure is a *merge tree*.

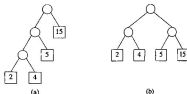


Figure 7.27: Possible two-way merges

In the first merge tree, we begin by merging the runs of size 2 and 4 to get one of size 6; next this is merged with the run of size 5 to get a run of size 11; finally this run of size 11 is merged with the run of size 15 to get the desired sorted run of size 26. When merging is done using the first merge tree, some records are involved in only one merge, and others are involved in up to three merges. In the second merge tree, each record is involved in exactly two merges. This corresponds to the strategy in which complete merge passes are repeatedly made over the data.

The number of merges that an individual record is involved in is given by the distance of the corresponding external node from the root. So, the records of the run with 15 records are involved in one merge when the first merge tree of Figure 7.27 is used and in two merges when the second tree is used. Since the time for a merge is linear in the number of records being merged, the total merge

time is obtained by summing the products of the run lengths and the distance from the root of the corresponding external nodes. This sum is called the *weighted external path length*. For the two trees of Figure 7.27, the respective weighted external path lengths are

$$2 \cdot 3 + 4 \cdot 3 + 5 \cdot 2 + 15 \cdot 1 = 43$$

and

$$2 \cdot 2 + 4 \cdot 2 + 2 \cdot 3 + 5 \cdot 2 + 15 \cdot 2 = 52$$

The cost of a k -way merge of n runs of length q_i , $1 \leq i \leq n$, is minimized by using a merge tree of degree k that has minimum weighted external path length. We shall consider the case $k = 2$ only. The discussion is easily generalized to the case $k > 2$ (see the exercises).

We briefly describe another application for binary trees with minimum weighted external path length. Suppose we wish to obtain an optimal set of codes for messages M_1, \dots, M_{n+1} . Each code is a binary string that will be used for transmission of the corresponding message. At the receiving end the code will be decoded using a *decode tree*. A decode tree is a binary tree in which external nodes represent messages. The binary bits in the code word for a message determine the branching needed at each level of the decode tree to reach the correct external node. For example, if we interpret a zero as a left branch and a one as a right branch, then the decode tree of Figure 7.28 corresponds to codes 000, 001, 01, and 1 for messages M_1, M_2, M_3 , and M_4 , respectively. These codes are called *Huffman codes*. The cost of decoding a code word is proportional to the number of bits in the code. This number is equal to the distance of the corresponding external node from the root node. If q_i is the relative frequency with which message M_i will be transmitted, then the expected decoding time is

$$\sum_{1 \leq i \leq n+1} q_i d_i$$

where d_i is the distance of the external node for message M_i from the root node. The expected decoding time is minimized by choosing code words resulting in a decode tree with minimal weighted external path length.

A very nice solution to the problem of finding a binary tree with minimum weighted external path length has been given by D. Huffman. This solution begins with a min heap of n single-node trees. The single node in each of these trees represents one of the provided q_i 's and its weight is q_i . Huffman's algorithm then repeatedly extracts two minimum-weight trees a and b from the min heap, combines them into a single binary tree c by creating a new root whose left and right subtrees are a and b respectively and whose weight is the sum of the weights of a and b , and inserts c into the min-heap. After $n - 1$ rounds of this extract, combine, insert process, the min heap will be left with a single binary



Figure 7.28: A decode tree

tree which is asserted to be a binary tree with minimum weighted external path length. Program 7.23 gives Huffman's algorithm as a C++ function. The template class *TreeNode* is defined in Chapter 5. We use the *data* field of a node to store the weight of the binary tree rooted at that node. It is assumed that the function *Huffman* is a friend of *TreeNode* as well as of *MinHeap*.

```

template <class T>
void Huffman(MinHeap<TreeNode<T>*> heap, int n)
// heap is initially a min heap of n single-node binary trees as described above.
for (int i = 0; i < n-1; i++)
    // combine two minimum-weight trees
    TreeNode<T> *first = heap.Pop();
    TreeNode<T> *second = heap.Pop();
    TreeNode<T> *bt = new BinaryTreeNode<T>(first, second,
                                             first->data + second->data);
    heap.Push(bt);
}
  
```

Program 7.23: Finding a binary tree with minimum weighted external path length

Example 7.15: Suppose we have the weights $q_1 = 2$, $q_2 = 3$, $q_3 = 5$, $q_4 = 7$, $q_5 = 9$, and $q_6 = 13$. Then the sequence of trees we would get is given in Figure

7.29 (the number in a circular node represents the sum of the weights of external nodes in that subtree).

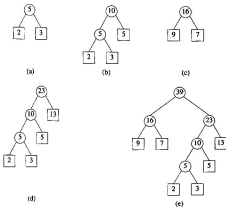


Figure 7.29: Construction of a Huffman tree

The weighted external path length of this tree is

$$2 \cdot 4 + 3 \cdot 4 + 5 \cdot 3 + 13 \cdot 2 + 7 \cdot 2 + 9 \cdot 2 = 93$$

By comparison, the best complete binary tree has weighted path length 95. \square

Analysis of Huffman: The main loop is executed $n - 1$ times. Each call to *Pop*

and *Push* requires $O(\log n)$ time. Hence, the asymptotic computing time is $O(n \log n)$. The correctness proof is left as an exercise. \square

EXERCISES

1. (a) n records are to be sorted on a computer with a memory capacity of S records ($S \ll n$). Assume that the entire S -record capacity may be used for input/output buffers. The input is on disk and consists of m runs. Assume that each time a disk access is made, the seek time is t_s and the latency time is t_l . The transmission time is t_t per record transmitted. What is the total input time for phase two of external sorting if a k -way merge is used with internal memory partitioned into input/output buffers to permit overlap of input, output, and CPU processing as in *Buffering* (Program 7.21)?
- (b) Let the CPU time needed to merge all the runs together be t_{CPU} (we may assume it is independent of k and hence constant). Let $t_s = 80 \text{ ms}$, $t_l = 20 \text{ ms}$, $n = 200,000$, $m = 64$, $t_t = 10^{-5}$ second, and $S = 2000$. Obtain a rough plot of the total input time, t_{input} , versus k . Will there always be a value of k for which $t_{CPU} = t_{input}$?
2. (a) Show that function *Huffman* (Program 7.23) correctly generates a binary tree of minimal weighted external path length.
- (b) When n runs are to be merged together using an m -way merge, Huffman's method can be generalized to the following rule: "First add $(1 - n) \bmod (m - 1)$ runs of length zero to the set of runs. Then, repeatedly merge the m shortest remaining runs until only one run is left." Show that this rule yields an optimal merge pattern for m -way merging.

7.11 REFERENCES AND SELECTED READINGS

A comprehensive discussion of sorting and searching may be found in *The Art of Computer Programming: Sorting and Searching*, by D. Knuth, Vol. 3, Second Edition, Addison-Wesley, Reading, MA, 1998.

CHAPTER 8

Hashing

8.1 INTRODUCTION

In this chapter, we again consider the ADT dictionary that was introduced in Chapter 5 (ADT 5.3). Examples of dictionaries are found in many applications, including the spelling checker, the thesaurus, the index for a database, and the symbol tables generated by loaders, assemblers, and compilers. When a dictionary with n entries is represented as a binary search tree as in Chapter 5, the dictionary operations *Get*, *Insert* and *Delete* take $O(n)$ time. These dictionary operations may be performed in $O(\log n)$ time using a balanced binary search tree (Chapter 10). In this chapter, we examine a technique, called hashing, that enables us to perform the dictionary operations *Get*, *Insert* and *Delete* in $O(1)$ expected time. We divide our discussion of hashing into two parts: *static hashing* and *dynamic hashing*.

8.2 STATIC HASHING

8.2.1 Hash Tables

In *static hashing* the dictionary pairs are stored in a table, h , called the *hash table*. The hash table is partitioned into b buckets, $h[0], \dots, h[b-1]$. Each bucket is capable of holding s dictionary pairs (or pointers to this many pairs). Thus, a bucket is said to consist of s slots, each slot being large enough to hold one dictionary pair. Usually $s = 1$, and each bucket can hold exactly one pair. The address or location of a pair whose key is k is determined by a hash function, h , which maps keys into buckets. Thus, for any key k , $h(k)$ is an integer in the range 0 through $b-1$. $h(k)$ is the *hash* or *home address* of k . Under ideal conditions, dictionary pairs are stored in their home buckets.

Definition: The *key density* of a hash table is the ratio n/T , where n is the number of pairs in the table and T is the total number of possible keys. The *loading density* or *loading factor* of a hash table is $\alpha = n/(sb)$. \square

Suppose our keys are at most six characters long, where a character may be a decimal digit or an uppercase letter, and that the first character is a letter. Then the number of possible keys is $T = \sum_{0 \leq i \leq 5} 26 \times 36^i > 1.6 \times 10^8$. Any reasonable

application, however, uses only a very small fraction of these. So, the key density, n/T , is usually very small. Consequently, the number of buckets, b , which is usually of the same magnitude as the number of keys, in the hash table is also much less than T . Therefore, the hash function h maps several different keys into the same bucket. Two keys, k_1 , and k_2 , are said to be *synonyms* with respect to h if $h(k_1) = h(k_2)$.

As indicated earlier, under ideal conditions, dictionary pairs are stored in their home buckets. Since many keys typically have the same home bucket, it is possible that the home bucket for a new dictionary pair is full at the time we wish to insert this pair into the dictionary. When this situation arises, we say that an *overflow* has occurred. A *collision* occurs when the home bucket for the new pair is not empty at the time of insertion. When each bucket has 1 slot (i.e., $s = 1$), collisions and overflows occur simultaneously.

Example 8.1: Consider a hash table with $b = 26$ buckets and $s = 2$. Assume that there are $n = 10$ distinct keys and that each key begins with a letter. The loading factor, α , for this table is $10/52 = 0.19$. The hash function h must map each of the possible keys into one of the numbers 0 to 25. If the internal binary representation for the letters A to Z corresponds to the numbers 0 to 25, respectively, then the function h defined by $h(k) =$ the first character of k will hash all

460 Hashing

keys into the hash table. The home buckets for GA, D, A, G, L, A2, A1, A3, A4, and E are 6, 3, 0, 6, 11, 0, 0, 0, 0, and 4, respectively. The keys A, A1, A2, A3, and A4 are synonyms. So also are G and GA. Figure 8.1 shows the keys GA, D, A, G, and A2 entered into the hash table. Though not shown in the figure, L is in slot 1 of bucket 11.

	Slot 1	Slot 2
0	A	A2
1		
2		
3	D	
4		
5		
6	GA	G
⋮	⋮	⋮
⋮	⋮	⋮
25		

Figure 8.1: Hash table with 26 buckets and two slots per bucket

Note that GA and G are in the same bucket and each bucket has two slots. Similarly, the synonyms A and A2 are in the same bucket. The next key, A1, hashes into bucket 0. This bucket is full and a search of the bucket indicates that A1 is not in the bucket. An overflow has now occurred. Where in the table should A1 be entered so that it may be retrieved when needed? □

When no overflows occur, the time required to insert, delete or search using hashing depends only on the time required to compute the hash function and the time to search one bucket. Hence, the insert, delete and search times are independent of n , the number of entries in the dictionary. Since the bucket size, s , is usually small (for internal-memory tables s is usually 1) the search within a bucket is carried out using a sequential search.

The hash function of Example 8.1 is not well suited for most practical applications because of the very large number of collisions and resulting overflows that occur. This is so because it is not unusual to find dictionaries in which many of the keys begin with the same letter. Ideally, we would like to choose a hash function that is both easy to compute and results in very few

collisions. Since the ratio b/T is usually very small, it is impossible to avoid collisions altogether.

In summary, hashing schemes use a hash function to map keys into hashable buckets. It is desirable to use a hash function that is both easy to compute and minimizes the number of collisions. Since the size of the key space is usually several orders of magnitude larger than the number of buckets and since the number of slots in a bucket is small, overflows necessarily occur. Hence, a mechanism to handle overflows is needed.

8.2.2 Hash Functions

A hash function maps a key into a bucket in the hash table. As mentioned earlier, the desired properties of such a function are that it be easy to compute and that it minimize the number of collisions. In addition, we would like the hash function to be such that it does not result in a biased use of the hash table for random inputs; that is, if k is a key chosen at random from the key space, then we want the probability that $h(k) = i$ to be $1/b$ for all buckets i . With this stipulation, a random key has an equal chance of hashing into any of the buckets. A hash function satisfying this property is called a *uniform hash function*.

Several kinds of uniform hash functions are in use in practice. Some of these compute the home bucket by performing arithmetic (e.g., multiplication and division) on the key. Since, in many applications, the data type of the key is not one for which arithmetic operations are defined (e.g., string), it is necessary to first convert the key into an integer (say) and then perform arithmetic on the obtained integer. In the following subsections, we describe four popular hash functions as well as ways to convert strings into integers.

8.2.2.1 Division

This hash function, which is the most widely used hash function in practice, assumes the keys are non-negative integers. The home bucket is obtained by using the modulo (%) operator. The key k is divided by some number D , and the remainder is used as the home bucket for k . More formally,

$$h(k) = k \% D$$

This function gives bucket addresses in the range 0 through $D - 1$, so the hash table must have at least $b = D$ buckets. Although for most key spaces, every choice of D makes h a uniform hash function, the number of overflows on real-world dictionaries is critically dependent on the choice of D . If D is divisible by two, then odd keys are mapped to odd buckets (as the remainder is odd), and

even keys are mapped to even buckets. Since real-world dictionaries tend to have a bias toward either odd or even keys, the use of an even divisor D results in a corresponding bias in the distribution of home buckets. In practice, it has been found that for real-world dictionaries, the distribution of home buckets is biased whenever D has small prime factors such as 2, 3, 5, 7 and so on. However, the degree of bias decreases as the smallest prime factor of D increases. Hence, for best performance over a variety of dictionaries, you should select D so that it is a prime number. With this selection, the smallest prime factor of D is D itself. For most practical dictionaries, a very uniform distribution of keys to buckets is seen even when we choose D such that it has no prime factor smaller than 10.

When you write a C++ hash table class for general use, the size of the dictionary to be accommodated in the hash table is not known. This makes it impractical to choose D as suggested above. So, we relax the requirement on D even further and require only that D be odd. In addition, we set b equal to the divisor D . As the size of the dictionary grows, it will be necessary to increase the size of the hash table A dynamically. To satisfy the relaxed requirement on D , array doubling results in increasing the number of buckets (and hence the divisor D) from b to $2b + 1$.

8.2.2.2 Mid-Square

The mid-square hash function determines the home bucket for a key by squaring the key and then using an appropriate number of bits from the middle of the square to obtain the bucket address; the key is assumed to be an integer. Since the middle bits of the square usually depend on all bits of the key, different keys are expected to result in different hash addresses with high probability, even when some of the digits are the same. The number of bits to be used to obtain the bucket address depends on the table size. If r bits are used, the range of values is 0 through $2^r - 1$. So the size of hash tables is chosen to be a power of two when the mid-square function is used.

8.2.2.3 Folding

In this method the key k is partitioned into several parts, all but possibly the last being of the same length. These partitions are then added together to obtain the hash address for k . There are two ways of carrying out this addition. In the first, all but the last partition are shifted so that the least significant bit of each lines up with the corresponding bit of the last partition. The different partitions are now added together to get $h(k)$. This method is known as *shift folding*. In the second method, *folding at the boundaries*, the key is folded at the partition boundaries.

and digits falling into the same position are added together to obtain $h(k)$. This is equivalent to reversing every other partition and then adding.

Example 8.2: Suppose that $k = 12320324111230$, and we partition it into parts that are three decimal digits long. The partitions are $P_1 = 123$, $P_2 = 203$, $P_3 = 241$, $P_4 = 112$, and $P_5 = 20$. Using shift folding, we obtain

$$h(k) = \sum_{i=1}^5 P_i = 123 + 203 + 241 + 112 + 20 = 699$$

When folding at the boundaries is used, we first reverse P_3 and P_4 to obtain 302 and 211, respectively. Next, the five partitions are added to obtain $h(k) = 123 + 302 + 241 + 211 + 20 = 897$. \square

8.2.2.4 Digit Analysis

This method is particularly useful in the case of a static file where all the keys in the table are known in advance. Each key is interpreted as a number using some radix r . The same radix is used for all the keys in the table. Using this radix, the digits of each key are examined. Digits having the most skewed distributions are deleted. Enough digits are deleted so that the number of remaining digits is small enough to give an address in the range of the hash table.

8.2.2.5 Converting Keys to Integers

To use some of the described hash functions, keys need to first be converted to nonnegative integers. Since all hash functions hash several keys into the same home bucket, it is not necessary for us to convert keys into unique nonnegative integers. It is ok for us to convert the strings *data*, *structures*, and *algorithms* into the same integer (say, 199). In this section, we consider only the conversion of strings into non-negative integers. Similar methods may be used to convert other data types into non-negative integers to which the described hash functions may be applied.

Example 8.3: [Converting Strings to Integers] Since it is not necessary to convert strings into unique nonnegative integers, we can map every string, no matter how long, into a 16-bit integer. Program 8.1 shows you one way to do this.

Program 8.1 converts pairs of characters into a unique integer and then sums these unique integers. Although it would have been easier to simply add all the characters together (rather than shift every other one by 8 bits), doing so would give us integers that are not much more than 8 bits long. For example,

```

unsigned int StringToInt(string s)
{
    // Convert s into a nonnegative int that depends on all characters of s.
    int length = (int) s.length(); // number of characters in s
    unsigned int answer = 0;
    if (length % 2 == 1)
    {
        // length is odd
        answer = s.at(length - 1);
        length--;
    }

    // length is now even
    for (int i = 0; i < length; i += 2)
    {
        // do two characters at a time
        answer += s.at(i);
        answer += ((int) s.at(i + 1)) << 8;
    }

    return answer;
}

```

Program 8.1: Converting a string into a non-negative integer

strings that are eight characters long would produce integers up to 11 bits long. Shifting by 8 bits allows us to cover the entire range of integers even with strings that are two characters long. □

The C++ STL provides specializations of the STL template class `hash<T>` that transform instances of type `T` into a nonnegative integer of type `size_t`. Program 8.2 shows a possible specialization of `hash<T>` for the case when `T` is the STL class `string`. This specialization is an alternative to the method of Program 8.1.

8.2.3 Secure Hash Functions

Hash functions with additional properties find several applications in computer security. One such application is *message authentication*. Consider a message M that is to be transmitted over an insecure channel from A to B . We want B to be confident that the received message is the original message that was transmitted, and not a forgery. Assume, for simplicity, that we have a means to transmit

```

template<>
class hash<string> {
public:
    size_t operator()(const string theKey) const
    { // Convert theKey to a nonnegative integer.
        unsigned long hashValue = 0;
        int length = (int) theKey.length();
        for (int i = 0; i < length; i++)
            hashValue = 5 * hashValue + theKey.at(i);

        return size_t(hashValue);
    }
};

```

Program 8.2: The specialization `hash<string>`

messages much smaller than M securely. For example, we may encrypt the smaller message or transmit the smaller message on a more expensive but far more secure channel than used for the transmission of a long message. One way to accomplish our task is to use a hash function h and transmit M 's hash value $h(M)$ using the more secure method; the message M is transmitted over the insecure channel as before. Suppose message M is altered along the insecure channel so that B receives a different message M' . B will now compute $h(M')$ and compare this with the $h(M)$ value it received. If $h(M)$ and $h(M')$ are different, B will know that it did not receive the correct message. Notice that if M and M' are different but synonyms, $h(M) = h(M')$ and B will incorrectly conclude that it has received the correct message. Therefore, in the cited application, we want to use a hash function h for which it is difficult for a malicious user who has knowledge of M and h to determine a synonym for M . This property is called *weak collision resistance*.

Other properties for secure hash functions that are useful in other scenarios are (1) *one-way property*: for a given c , it is computationally difficult to find a k such that $h(k)=c$ and (2) *strong collision resistance*: it is computationally difficult to find a pair (x,y) such that $h(x)=h(y)$.

Several cryptographic hash functions with these properties have been developed. In addition to the security-related properties mentioned earlier, these hash functions have additional features designed to make them practical: (1) h can be applied to a block of data of any size, (2) h produces a fixed-length hash code, and (3) $h(k)$ is relatively easy to compute for any given k , making both hardware and software implementations practical.

The Secure Hash Algorithm (SHA) was developed at the National Institute of Standards and Technology (NIST) in the USA. We describe the SHA-1 function. The input to SHA is any message with maximum length less than 2^{64} bits. Its output is a 160-bit code. An outline of the algorithm is shown in Program 8.3.

Step 1: Preprocess the message so that its length is $q \cdot 512$ bits for some integer q . The preprocessing may entail adding a string of zeroes to the end of the message.

Step 2: Initialize the 160-bit output buffer OB , which comprises five 32-bit registers A , B , C , D , and E , with $A = 67452301$, $B = \text{efcdab89}$, $C = 98badcfe$, $D = 10325476$, $E = \text{c3d2e1f0}$ (all values are in hexadecimal).

Step 3:

```

for (int i = 1; i <= q; i++) {
    Let  $B_i$  be  $i$ th block of 512 bits of the message;
     $OB = F(OB, B_i)$ ;
}

```

Step 4: Output OB .

Program 8.3: SHA algorithm

The function F in Step 3 itself consists of four rounds of 20 steps each! Figure 8.2 shows the form of each of these 80 steps.

Example 8.4: Assume that the contents of registers A through E are specified as above; assume f is $(B \oplus C \oplus D)$, $K_0 = 3a827999$, and $W_0 = 00000000$. Note that \oplus is the exclusive-or operator. Let us compute the updated values of the registers A through E .

We have $B = 67452301$ (the original value in A), $D = 98badcfe$ (the original value in C), and $E = 10325476$ (the original value in D). C is obtained by a circular left-shift of B by 30 bits (which is equivalent to a circular right-shift by 2 bits). Since, $B = 1110\ 1111\ 1100\ 1101\ 1010\ 1011\ 1000\ 1001$, the updated C is $0111\ 1011\ 1111\ 0011\ 0110\ 1010\ 1110\ 0010 = 7bf36ae2$.

We finally compute the new value in A by adding together five quantities. Of these, E , W_0 , and K_0 are given, $f(B, C, D) = B \oplus C \oplus D = 67452301$, and $S^2(A) = \text{efa4602c}$. Adding (mod 2^{32}) gives $6e\ 3e\ de\ b6$. \square

Example 8.5: The number of times the atomic SHA operation is executed for a 1KB message is determined as follows. Since 1KB = 1024 bytes = $16 \cdot 512$ bits, the preprocessing Step 1, doesn't add bits to the message and $q = 16$ in Program 8.3. Hence, the for loop of Step 3 of the SHA algorithm is iterated 16 times. Each iteration consists of 80 atomic SHA ops, so the number of atomic

random probing. In linear probing, when inserting a new pair whose key is k , we search the hash table buckets in the order, $h[A(k) + i] \% b$, $0 \leq i \leq b - 1$, where h is the hash function and b is the number of buckets. This search terminates when we reach the first unfilled bucket and the new pair is inserted into this bucket. In case no such bucket is found, the hash table is full and it is necessary to increase the table size. In practice, to ensure good performance, table size is increased when the loading density exceeds a prespecified threshold such as 0.75 rather than when the table is full. Notice that when we resize the hash table, we must change the hash function as well. For example, when the division hash function is used, the divisor equals the number of buckets. This change in the hash function potentially changes the home bucket for each key in the hash table. So, all dictionary entries need to be remapped into the new larger table.

Example 8.6: Assume we have a 26-bucket table with one slot per bucket and the following keys: GA, D, A, G, L, A2, A1, A3, A4, Z, ZA, E. For simplicity we choose the hash function $h(k) = \text{first character of } k$. Initially, all entries in the table are null. Since $h(GA) = 6$, and $h[6]$ is empty, GA is entered into $h[6]$. D and A get entered into the buckets $h[3]$ and $h[0]$, respectively. The next key G has $h(G) = 6$. This bucket is already used by GA. The next vacant slot is $h[7]$, so G is entered there. L enters $h[11]$. A2 collides with A at $h[0]$, the bucket overflows, and A2 is entered at the next vacant slot, $h[1]$. A1, A3, and A4 are entered at $h[2]$, $h[4]$, and $h[5]$, respectively. Z is entered at $h[25]$, ZA at $h[8]$ (the hash table is used circularly), and E collides with A3 at $h[4]$ and is eventually entered at $h[9]$. Figure 8.3 shows the resulting table. □

When $i = 1$ and linear probing is used to handle overflows, a hash table search for the pair with key k proceeds as follows:

- (1) Compute $h(k)$.
- (2) Examine the hash table buckets in the order $h[h(k)]$, $h[(h(k) + 1) \% b]$, \dots , $h[(h(k) + j) \% b]$ until one of the following happens:
 - (a) The bucket $h[(h(k) + j) \% b]$ has a pair whose key is k ; in this case, the desired pair has been found.
 - (b) $h[(h(k) + j) \% b]$ is empty; k is not in the table.
 - (c) We return to the starting position $h[h(k)]$; the table is full and k is not in the table.

Program 8.4 is the resulting search function. This function assumes that the hash table h stores pointers to dictionary pairs.

In Example 8.6 we saw that when linear probing is used to resolve overflows, keys tend to cluster together. Moreover, adjacent clusters tend to

0	A
1	A ²
2	A ¹
3	D
4	A ³
5	A ⁴
6	GA
7	G
8	ZA
9	E
10	
11	L
12	
13	
	⋮
24	
25	Z

Figure 8.3: Hash table with linear probing (26 buckets, one slot per bucket)

coalesce, thus increasing the search time. To locate the key ZA in the table of Figure 8.3, it is necessary to examine $h[25]$, $h[0]$, \dots , $h[8]$ — a total of 10 comparisons. This is far worse than the worst-case behavior for the tree tables we shall see in Chapter 10. If each of the keys in the table of Figure 8.3 is retrieved exactly once, then the number of buckets examined is 1 for A, 2 for A², 3 for A¹, 1 for D, 5 for A³, 6 for A⁴, 1 for GA, 2 for G, 10 for ZA, 6 for E, 1 for L, and 1 for Z — for a total of 39 buckets examined. The average number of buckets examined in a successful search of our example hash table is 3.25.

When linear probing is used together with a uniform hash hash function, the expected average number of key comparisons to look up a key is approximately $(2 - \alpha)/2 = 2\alpha$, where α is the loading density. This is the average over all possible sets of keys yielding the given loading density and using a uniform function h . In Example 8.6, $\alpha = 12/26 = .47$ and the average number of key comparisons is 1.5. Even though the average number of comparisons is small, the worst case can be quite large.

```

template <class K, class E>
pair<K, E> LinearProbing <K, E>::Get(const K& k)
{
    // Search the linear probing hash table ht (each bucket has exactly one slot) for k.
    // If a pair with this key is found, return a pointer to this pair; otherwise, return 0.
    int i = h(k); // home bucket
    int j;
    for (j = i; ht[j] && ht[j] != k; j++) {
        j = (j + 1) % b; // treat the table as circular
        if (j == i) return 0; // back to start point
    }
    if (ht[j] == k) return ht[j];
    return 0;
}

```

Program 8.4: Linear probing

Some improvement in the growth of clusters and hence in the average number of probes needed for searching can be obtained by *quadratic probing*. Linear probing was characterized by searching the buckets $(h(k) + i) \% b$, $0 \leq i \leq b - 1$, where b is the number of buckets in the table. In quadratic probing, a quadratic function of i is used as the increment. In particular, the search is carried out by examining buckets $h(k)$, $(h(k) + i^2) \% b$, and $(h(k) - i^2) \% b$ for $1 \leq i \leq (b - 1)/2$. When b is a prime number of the form $4j + 3$, for j an integer, the quadratic search described above examines every bucket in the table. Figure 8.4 lists some primes of the form $4j + 3$.

Prime	j	Prime	j
3	0	43	10
7	1	59	14
11	2	127	31
19	4	251	62
23	5	503	125
31	7	1019	254

Figure 8.4: Some primes of the form $4j + 3$

An alternative method to retard the growth of clusters is to use a series of hash functions h_1, h_2, \dots, h_n . This method is known as *rehashing*. Buckets

$h_i(k)$, $1 \leq i \leq m$ are examined in that order. Yet another alternative, random probing, is explored in the exercises.

8.3.4.2 Chaining

Linear probing and its variations perform poorly because the search for a key involves comparison with keys that have different hash values. In the hash table of Figure 8.3, for instance, searching for the key ZA involves comparisons with the buckets $h[0]$ to $h[7]$, even though none of the keys in these buckets had a collision with $h[25]$ and so cannot possibly be ZA. Many of the comparisons can be saved if we maintain lists of keys, one list per bucket, each list containing all the synonyms for that bucket. If this is done, a search involves computing the hash address $h(k)$ and examining only those keys in the list for $h(k)$. Although the list for $h(k)$ may be maintained using any data structure that supports the search, insert and delete operations (e.g., arrays, chains, search trees), chains are most frequently used. We typically use an array $h[0:b-1]$ of type *ChainNode* $\langle \text{pair} \langle K, E \rangle \rangle^*$ with $h[i]$ pointing to the first node of the chain for bucket i . Program 8.5 gives the search algorithm for chained hash tables.

```

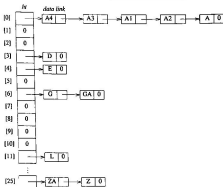
template <class K, class E>
pair <K, E>* Chaining <K, E>::Get(const K& k)
{ // Search the chained hash table h for k.
  // If a pair with this key is found, return a pointer to this pair, otherwise, return 0.
  int i = h(k); // home bucket
  // search the chain h[i]
  for (ChainNode <pair <K, E>>* current = h[i]; current;
      current = current->link)
    If (current->data.first == k) return &current->data;
  return 0;
}

```

Program 8.5: Chain search

When chaining is used on the data of Example 8.6, the hash chains of Figure 8.5 are obtained. To insert a new key, k , into a chain, we must first verify that it is not currently on the chain. Following this, k may be inserted at any position of the chain. In the example of Figure 8.5, new keys were inserted at the front of the chains. Deletion from a chained hash table can be done by removing the appropriate node from its chain.

The number of comparisons needed to search for any of the keys in the



Hash table with 26 buckets.

Figure 8.5: Hash chains corresponding to Figure 8.3

chained hash table of Figure 8.5 is one for each of A4, D, E, G, L, and ZA; two for each of A3, GA, and Z; three for A1; four for A2; and five for A, for a total of 24. The average is now two, which is considerably less than the average for linear probing.

When chaining is used along with a uniform hash function, the expected average number of key comparisons for a successful search is $1 + \alpha/2$, where α is the loading density n/b (b = number of buckets). For $\alpha = 0.5$ this number is 1.25, and for $\alpha = 1$ it is 1.5. The corresponding numbers for linear probing are 1.5 and b , the table size.

The performance results cited in this section tend to imply that provided we use a uniform hash function, performance depends only on the method used to

handle overflows. Although this is true when the keys are selected at random from the key space, it is not true in practice. In practice, there is a tendency to make a biased use of keys. Hence, in practice, different hash functions result in different performance. Generally, the division hash function coupled with chaining yields best performance.

The worst-case number of comparisons needed for a successful search remains $O(n)$ regardless of whether we use open addressing or chaining. The worst-case number of comparisons may be reduced to $O(\log n)$ by storing synonyms in a balanced search tree (see Chapter 10) rather than in a chain.

8.2.5 Theoretical Evaluation of Overflow Techniques

The experimental evaluation of hashing techniques indicates a very good performance over conventional techniques such as balanced trees. The worst-case performance for hashing can, however, be very bad. In the worst case, an insertion or a search in a hash table with n keys may take $O(n)$ time. In this section, we present a probabilistic analysis for the expected performance of the chaining method and state without proof the results of similar analyses for the other overflow handling methods. First, we formalize what we mean by expected performance.

Let $h: [0: b-1]$ be a hash table with b buckets, each bucket having one slot. Let h be a uniform hash function with range $[0, b-1]$. If n keys k_1, k_2, \dots, k_n are entered into the hash table, then there are b^n distinct hash sequences $h(k_1), h(k_2), \dots, h(k_n)$. Assume that each of these is equally likely to occur. Let S_n denote the expected number of key comparisons needed to locate a randomly chosen k_i , $1 \leq i \leq n$. Then, S_n is the average number of comparisons needed to find the j th key k_j , averaged over $1 \leq j \leq n$, with each j equally likely, and averaged over all b^n hash sequences, assuming each of these also to be equally likely. Let U_n be the expected number of key comparisons when a search is made for a key not in the hash table. This hash table contains n keys. The quantity U_n may be defined in a manner analogous to that used for S_n .

Theorem 8.1: Let $\alpha = n/b$ be the loading density of a hash table using a uniform hashing function h . Then

(1) for linear open addressing

$$U_n = \frac{1}{2} \left(1 + \frac{1}{(1-\alpha)^2} \right)$$

$$S_n = \frac{1}{2} \left(1 + \frac{1}{1-\alpha} \right)$$

474 Hashing

- (2) for rehashing, random probing, and quadratic probing

$$U_n = 1/(1-\alpha)$$

$$S_n = - \left\lceil \frac{1}{\alpha} \right\rceil \log_e(1-\alpha)$$

- (3) for chaining

$$U_n = \alpha$$

$$S_n = 1 + \alpha/2$$

Proof: Exact derivations of U_n and S_n are fairly involved and can be found in Knuth's book *The Art of Computer Programming: Sorting and Searching* (see the References and Selected Readings section). Here we present a derivation of the approximate formulas for chaining. First, we must make clear our count for U_n and S_n . If the key k being sought has $h(k) = i$, and chain i has q nodes on it, then q comparisons are needed if k is not on the chain. If k is in the j th node of the chain, $1 \leq j \leq q$, then j comparisons are needed.

When the n keys are distributed uniformly over the b possible chains, the expected number in each chain is $n/b = \alpha$. Since U_n equals the expected number of keys on a chain, we get $U_n = \alpha$.

When the i th key, k_i , is being entered into the table, the expected number of keys on any chain is $(i-1)/b$. Hence, the expected number of comparisons needed to search for k_i after all n keys have been entered is $1 + (i-1)/b$ (this assumes that new entries will be made at the end of the chain). Thus,

$$S_n = \frac{1}{n} \sum_{i=1}^n [1 + (i-1)/b] = 1 + \frac{n-1}{2b} = 1 + \frac{\alpha}{2} \quad \square$$

EXERCISES

1. Show that the hash function $h(k) = k \% 17$ does not satisfy the one-way property, weak collision resistance, or strong collision resistance.
2. Consider a hash function $h(k) = k \% D$, where D is not given. We want to figure out what value of D is being used. We wish to achieve this using as few attempts as possible, where an attempt consists of supplying the function with k and observing $h(k)$. Indicate how this may be achieved in the following two cases.
 - (a) D is known to be a prime number in the range $[10, 20]$.
 - (b) D is of the form 2^k , where k is an integer in $[1, 5]$.

3. Write a C++ function to delete the pair with key k from a hash table that uses linear probing. Show that simply setting the slot previously occupied by the deleted pair to empty does not solve the problem. How must `Get` (Program 8.4) be modified so that a correct search is made in the situation when deletions are permitted? Where can a new key be inserted?
4. (a) Show that if quadratic searching is carried out in the sequence $(h(k) + q^2)$, $(h(k) + (q - 1)^2)$, \dots , $(h(k) + 1)$, $h(k)$, $(h(k) - 1)$, \dots , $(h(k) - q^2)$ with $q = (b - 1)/2$, then the address difference $\% b$ between successive buckets being examined is

$$b - 2, b - 4, b - 6, \dots, 5, 3, 1, 1, 3, 5, \dots, b - 6, b - 4, b - 2$$
- (b) Write a C++ function to search a hash table `ht` of size b for the key k . Use h as the hash function and the quadratic probing scheme discussed in the text to resolve overflows. Use the results of part (a) to reduce the computations.
5. [Morris 1968] In random probing, the search for a key, k , in a hash table with b buckets is carried out by examining the buckets in the order $h(k)$, $(h(k) + s(i)) \% b$, $1 \leq i \leq b - 1$ where $s(i)$ is a pseudo random number. The random number generator must satisfy the property that every number from 1 to $b - 1$ must be generated exactly once as i ranges from 1 to $b - 1$.
 - (a) Show that for a table of size 2^r , the following sequence of computations generates numbers with this property:
 Initialize q to 1 each time the search routine is called.
 On successive calls for a random number do the following:
 $q \leftarrow 5$
 $q = \text{low order } r + 2 \text{ bits of } q$
 $s(i) = q / 4$
 - (b) Write search and insert functions for a hash table using random probing and the mid-square hash function. Use the random number generator of (a).

It can be shown that for this method, the expected value for the average number of comparisons needed to search for a dictionary pair is $-(1/\alpha) \log(1 - \alpha)$ for large tables (α is the loading factor).

6. Develop a C++ hash table class in which overflows are resolved using a binary search tree. Use the division hash function with an odd divisor D and array doubling whenever the loading density exceeds a prespecified amount. Recall that in this context, array doubling actually increases the size of the hash table from its current size $b = D$ to $2b + 1$.

7. Write a C++ function to list all the keys in a hash table in lexicographic order. Assume that linear probing is used. How much time does your function take?
8. Let the binary representation of key k be $k_1k_2\ldots$. Let $|x|$ denote the number of bits in x and let the first bit of k_1 be 1. Let $|k_d| = \lceil |k|/2 \rceil$ and $|k_d| = \lfloor |k|/2 \rfloor$. Consider the following hash function

$$h(k) = \text{middle } q \text{ bits of } (k_1 \oplus k_2)$$

where \oplus is the exclusive-or operator. Is this a uniform hash function if keys are drawn at random from the space of integers? What can you say about the behavior of this hash function in actual dictionary usage?

9. [T. Gonzalez] Design a dictionary representation that allows you to search, insert, and delete in $O(1)$ time. Assume that the keys are integer and in the range $[0, m)$ and that $m + n$ units of space are available, where n is the number of insertions to be made. (Hint: Use two arrays, $a[n]$ and $b[m]$, where $a[i]$ will be the $(i+1)$ th pair inserted into the table. If k is the i th key inserted, then $b[k] = i$.) Write C++ functions to search, insert, and delete. Note that you cannot initialize the arrays a and b as this would take $O(n + m)$ time.
10. [T. Gonzalez] Let $s = \{s_1, s_2, \dots, s_n\}$ and $t = \{t_1, t_2, \dots, t_r\}$ be two sets. Assume $1 \leq s_i \leq m$, $1 \leq t_i \leq m$, and $1 \leq i_1 \leq m$, $1 \leq i \leq r$. Using the idea of Exercise 9, write a function to determine if $s \subseteq t$. Your function should work in $O(r + n)$ time. Since $s = t$ iff $s \subseteq t$ and $t \subseteq s$, one can determine in linear time whether two sets are the same. How much space is needed by your function?
11. [T. Gonzalez] Using the idea of Exercise 9, write an $O(n + m)$ time function to carry out the task of *Verify3* (Program 7.3). How much space does your function need?
12. Using the notation of Section 8.2.4, show that when linear probing is used

$$S_n = \frac{1}{n} \sum_{i=0}^{n-1} U_i$$

Using this equation and the approximate equality

$$U_n = \frac{1}{2} \left(1 + \frac{1}{(1 - \alpha)^2} \right) \quad \text{where } \alpha = \frac{n}{b}$$

show that

$$S_n = \frac{1}{2} \left(1 + \frac{1}{(1 - \alpha)} \right)$$

8.3 DYNAMIC HASHING

8.3.1 Motivation for Dynamic Hashing

To ensure good performance, it is necessary to increase the size of a hash table whenever its loading density exceeds a prespecified threshold. So, for example, if we currently have b buckets in our hash table and are using the division hash function with divisor $D = b$, then, when an insert causes the loading density to exceed the prespecified threshold, we use array doubling to increase the number of buckets to $2b + 1$. At the same time, the hash function divisor changes to $2b + 1$. This change in divisor requires us to rebuild the hash table by collecting all dictionary pairs in the original smaller size table and reinserting these into the new larger table. We cannot simply copy dictionary entries from the smaller table into corresponding buckets of the bigger table as the home bucket for each entry has potentially changed. For very large dictionaries that must be accessible on a 24/7 basis, the required rebuild means that dictionary operations must be suspended for unacceptably long periods while the rebuild is in progress. Dynamic hashing, which also is known as extendible hashing, aims to reduce the rebuild time by ensuring that each rebuild changes the home bucket for the entries in only 1 bucket. In other words, although table doubling increases the total time for a sequence of n dictionary operations by only $O(n)$, the time required to complete an insert that triggers the doubling is excessive in the context of a large dictionary that is required to respond quickly on a per operation basis. The objective of dynamic hashing is to provide acceptable hash table performance on a per operation basis.

We consider two forms of dynamic hashing—one uses a directory and the other does not—in this section. For both forms, we use a hash function h that maps keys into non-negative integers. The range of h is assumed to be sufficiently large and we use $h(k, p)$ to denote the integer formed by the p least significant bits of $h(k)$.

For the examples of this section, we use a hash function $h(k)$ that transforms keys into 6-bit non-negative integers. Our example keys will be two characters each and h transforms letters such as A, B and C into the bit sequence 100, 101, and 110, respectively. Digits 0 through 7 are transformed into their 3-bit representation. Figure 8.6 shows 8 possible 2 character keys together with the binary representation of $h(k)$ for each. For our example hash function, $h(A0.1) = 0$, $h(A.1.3) = 1$, $h(B.1.4) = 1001 = 9$, and $h(C.1.6) = 110.001 = 49$.

k	$h(k)$
A0	100 000
A1	100 001
B0	101 000
B1	101 001
C1	110 001
C2	110 010
C3	110 011
C5	110 101

Figure 8.6: An example hash function

8.3.2 Dynamic Hashing Using Directories

We employ a directory, d , of pointers to buckets. The size of the directory depends on the number of bits of $h(k)$ used to index into the directory. When indexing is done using, say, $h(k, 2)$, the directory size is $2^2 = 4$; when $h(k, 5)$ is used, the directory size is 32. The number of bits of $h(k)$ used to index the directory is called the *directory depth*. The size of the directory is 2^r , where r is the directory depth and the number of buckets is at most equal to the directory size. Figure 8.7 (a) shows a dynamic hash table that contains the keys A0, B0, A1, B1, C1, and C3. This hash table uses a directory whose depth is 2 and uses buckets that have 2 slots each. In Figure 8.7, the directory is shaded while the buckets are not. In practice, the bucket size is often chosen to match some physical characteristic of the storage media. For example, when the dictionary pairs reside on disk, a bucket may correspond to a disk track or sector.

To search for a key k , we merely examine the bucket pointed to by $d[h(k, r)]$, where r is the directory depth.

Suppose we insert C5 into the hash table of Figure 8.7 (a). Since, $h(C5, 2) = 01$, we follow the pointer, $d[01]$, in position 01 of the directory. This gets us to the bucket with A1 and B1. This bucket is full and we get a bucket overflow. To resolve the overflow, we determine the least u such that $h(k, u)$ is not the same for all keys in the overflowed bucket. In case the least u is greater than the directory depth, we increase the directory depth to this least u value. This requires us to increase the directory size but not the number of buckets. When the directory size doubles, the pointers in the original directory are duplicated so that the pointers in each half of the directory are the same. A quadrupling of the directory size may be handled as two doublings and so on. For our example, the least

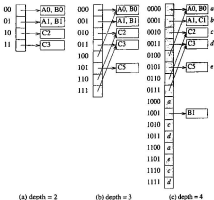


Figure 8.7: Dynamic hash tables with directories

u for which $h(k, u)$ is not the same for A1, B1, and C5 is 3. So, the directory is expanded to have depth 3 and size 8. Following the expansion, $d[i] = d[i + 4]$, $0 \leq i < 4$.

Following the resizing (if any) of the directory, we split the overflowed bucket using $h(k, u)$. In our case, the overflowed bucket is split using $h(k, 3)$. For A1 and B1, $h(k, 3) = 001$ and for C5, $h(k, 3) = 101$. So, we create a new bucket with C5 and place a pointer to this bucket in $d[101]$. Figure 8.7 (b) shows the result. Notice that each dictionary entry is in the bucket pointed at by the directory position $h(k, 3)$, although, in some cases the dictionary entry is also pointed at by other buckets. For example, bucket 100 also points to A0 and B0, even

though $h(A0,3) = h(B0,3) \neq 000$.

Suppose that instead of C5, we were to insert C1. The pointer in position $h(C1,2) = 01$ of the directory of Figure 8.7 (a) gets us to the same bucket as when we were inserting C5. This bucket overflows. The least u for which $h(k,u)$ isn't the same for A1, B1 and C1 is 4. So, the new directory depth is 4 and its new size is 16. The directory size is quadrupled and the pointers $d[0:3]$ are replicated 3 times to fill the new directory. When the overflowed bucket is split, A1 and C1 are placed into a bucket that is pointed at by $d[0001]$ and B1 into a bucket pointed at by $d[1001]$.

When the current directory depth is greater than or equal to u , some of the other pointers to the split bucket also must be updated to point to the new bucket. Specifically, the pointers in positions that agree with the last u bits of the new bucket need to be updated. The following example illustrates this. Consider inserting A4 ($h(A4) = 100\ 100$) into Figure 8.7 (b). Bucket $d[100]$ overflows. The least u is 3, which equals the directory depth. So, the size of the directory is not changed. Using $h(k,3)$, A0 and B0 hash to 000 while A4 hashes to 100. So, we create a new bucket for A4 and set $d[100]$ to point to this new bucket.

As a final insert example, consider inserting C1 into Figure 8.7 (b). $h(C1,3) = 001$. This time, bucket $d[001]$ overflows. The minimum u is 4 and so it is necessary to double the directory size and increase the directory depth to 4. When the directory is doubled, we replicate the pointers in the first half into the second half. Next we split the overflowed bucket using $h(k,4)$. Since $h(k,4) = 0001$ for A1 and C1 and 1001 for B1, we create a new bucket with B1 and put C1 into the slot previously occupied by B1. A pointer to the new bucket is placed in $d[1001]$. Figure 8.7 (c) shows the resulting configuration. For clarity, several of the bucket pointers have been replaced by lowercase letters indicating the bucket pointed to.

Deletion from a dynamic hash table with a directory is similar to insertion. Although dynamic hashing employs array doubling, the time for this array doubling is considerably less than that for the array doubling used in static hashing. This is so because, in dynamic hashing, we need to rehash only the entries in the bucket that overflows rather than all entries in the table. Further, savings result when the directory resides in memory while the buckets are on disk. A search requires only 1 disk access; an insert makes 1 read and 2 write accesses to the disk; the array doubling requires no disk access.

8.3.3 Directoryless Dynamic Hashing

As the name suggests, in this method, we dispense with the directory, d , of bucket pointers used in the method of Section 8.3.2. Instead, an array, ht , of buckets is used. We assume that this array is as large as possible and so there is

no possibility of increasing its size dynamically. To avoid initializing such a large array, we use two variables q and r , $0 \leq q < 2^r$, to keep track of the active buckets. At any time, only buckets 0 through $2^r + q - 1$ are active. Each active bucket is the start of a chain of buckets. The remaining buckets on a chain are called *overflow buckets*. The active buckets 0 through $q - 1$ as well as the active buckets 2^r through $2^r + q - 1$ are indexed using $h(k, r + 1)$ while the remaining active buckets are indexed using $h(k, r)$. Each dictionary pair is either in an active or an overflow bucket.

Figure 8.8 (a) shows a directoryless hash table h with $r = 2$ and $q = 0$. The hash function is that of Figure 8.6, $h(B4) = 101\ 100$, and $h(B5) = 101\ 101$. The number of active buckets is 4 (indexed 00, 01, 10, and 11). The index of an active bucket identifies its chain. Each active bucket has 2 slots and bucket 00 contains B4 and A0. There are 4 bucket chains, each chain begins at one of the 4 active buckets and comprises only that active bucket (i.e., there are no overflow buckets). In Figure 8.8 (a), all keys have been mapped into chains using $h(k, 2)$. In Figure 8.8 (b), $r = 2$ and $q = 1$; $h(k, 3)$ has been used for chains 000 and 100 while $h(k, 2)$ has been used for chains 001, 010, and 011. Chain 001 has an overflow bucket; the capacity of an overflow bucket may or may not be the same as that of an active bucket.

To search for k , we first compute $h(k, r)$. If $h(k, r) < q$, then k , if present, is in a chain indexed using $h(k, r + 1)$. Otherwise, the chain to examine is given by $h(k, r)$. Program 8.6 gives the algorithm to search a directoryless dynamic hash table.

```

if ( $h(k, r) < q$ ) search the chain that begins at bucket  $h(k, r + 1)$ ;
else search the chain that begins at bucket  $h(k, r)$ ;

```

Program 8.6: Searching a directoryless hash table

To insert C5 into the table of Figure 8.8 (a), we use the search algorithm of Program 8.6 to determine whether or not C5 is in the table already. Chain 01 is examined and we verify that C5 is not present. Since the active bucket for the searched chain is full, we get an overflow. An overflow is handled by activating bucket $2^r + q$; reallocating the entries in the chain q between q and the newly activated bucket (or chain) $2^r + q$, and incrementing q by 1. In case q now becomes 2^r , we increment r by 1 and reset q to 0. The reallocation is done using $h(k, r + 1)$. Finally, the new pair is inserted into the chain where it would be searched for by Program 8.6 using the new r and q values.

For our example, bucket $4 = 100$ is activated and the entries in chain 00 ($q = 0$) are rehashed using $r + 1 = 3$ bits. B4 hashes to the new bucket 100 and A0 to bucket 000. Following this, $q = 1$ and $r = 2$. A search for C5 would examine

482 Hashing

chain 1 and so C5 is added to this chain using an overflow bucket (see Figure 8.8 (b)). Notice that at this time, the keys in buckets 001, 010 and 011 are hashed using $h(k, 2)$ while those in buckets 000 and 100 are hashed using $h(k, 3)$.

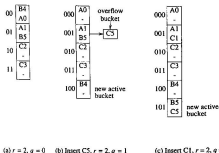


Figure 8.8: Inserting into a directoryless dynamic hash table

Let us now insert C1 into the table of Figure 8.8 (b). Since, $h(C1, 2) = 01 = q$, chain 01 = 1 is examined by our search algorithm (Program 8.6). The search verifies that C1 is not in the dictionary. Since the active bucket 01 is full, we get an overflow. We activate bucket $2^r + q = 5 = 101$ and rehash the keys A1, B5, and C3 that are in chain q . The rehashing is done using 3 bits. A1 is hashed into bucket 001 while B5 and C3 hash into bucket 101. q is incremented by 1 and the new key C1 is inserted into bucket 001. Figure 8.8 (c) shows the result.

EXERCISES

1. Write an algorithm to insert a dictionary pair into a dynamic hash table that uses a directory.
2. Write an algorithm to delete a dictionary pair from a dynamic hash table that uses a directory.
3. Write an algorithm to insert a dictionary pair into a directoryless dynamic hash table.
4. Write an algorithm to delete a dictionary pair from a directoryless dynamic hash table.

8.4 BLOOM FILTERS

8.4.1 An Application—Differential Files

Consider an application where we are maintaining an indexed file. For simplicity, assume that there is only one index and hence just a single key. Further assume that this is a dense index (i.e., one that has an entry for each record in the file) and that updates to the file (inserts, deletes, and changes to an existing record) are permitted. It is necessary to keep a backup copy of the index and file so that we can recover from accidental loss or failure of the working copy. This loss or failure may occur for a variety of reasons, which include corruption of the working copy due to a malfunction of the hardware or software. We shall refer to the working copies of the index and file as the *master index* and *master file*, respectively.

Since updates to the file and index are permitted, the backup copies of these generally differ from the working copies at the time of failure. So, it is possible to recover from the failure only if, in addition to the backup copies, we have a log of all updates made since the backup copies were created. We shall call this log the *transaction log*. To recover from the failure, it is necessary to process the backup copies and the transaction log to reproduce an index and file that correspond to the working copies at the time of failure. The time needed to recover is therefore a function of the sizes of the backup index and file and the size of the transaction log. The recovery time can be reduced by making more frequent backups. This results in a smaller transaction log. Making sufficiently frequent backups of the master index and file is not practical when these are very large and when the update rate is very high.

When only the file (but not the index) is very large, a reduction in the recovery time may be obtained by keeping updated records in a separate file called the *differential file*. The master file is unchanged. The master index is, however, changed to reflect the position of the most current version of the record with a given key. We assume that the addresses for differential-file records and

master-file records are different. As a result, by examining the address obtained from a search of the master index, we can tell whether the most current version of the record we are seeking is in the master file or in the differential file. The steps to follow when accessing a record with a given key are given in Program 8.7(b). Program 8.7(a) gives the steps when a differential file is not used.

Notice that when a differential file is used, the backup file is an exact replica of the master file. Hence, it is necessary to backup only the master index and differential file frequently. Since these are relatively small, it is feasible to do this. To recover from a failure of the master index or differential file, the transactions in the transaction log need to be processed using the backup copies of the master file, index, and differential file. The transaction log can be expected to be relatively small, so backups are done more frequently. To recover from a failure of the master file, we need merely make a new copy of its backup. When the differential file becomes too large, it is necessary to create a new version of the master file by merging the old master file and the differential file. This also results in a new index and an empty differential file. It is interesting to note that using a differential file as suggested does not affect the number of disk accesses needed to perform a file operation (see Program 8.7(a,b)).

Suppose that both the index and the file are very large. In this case the differential-file scheme discussed above does not work as well, as it is not feasible to backup the master index as frequently as is necessary to keep the transaction log sufficiently small. We can get around this difficulty by using a differential file and a differential index. The master index and master file remain unchanged as updates are performed. The differential file contains all newly inserted records and the current versions of all changed records. The differential index is an index to the differential file. This also has null address entries for deleted records. The steps needed to perform a file operation when both a differential index and file are used are given in Program 8.7(c). Comparing with Program 8.7(a), we see that additional disk accesses are frequently needed, as we will often first query the differential index and then the master index. Observe that the differential file is much smaller than the master file, so most requests are satisfied from the master file.

When a differential index and file are used, we must backup both of these with high frequency. This is possible, as both are relatively small. To recover from a loss of the differential index or file, we need to process the transactions in the transaction log using the available backup copies. To recover from a loss of the master index or master file, a copy of the appropriate backup needs to be made. When the differential index and/or file becomes too large, the master index and/or file is reorganized so that the differential index and/or file becomes empty.

Step 1:	Search master index for record address.
Step 2:	Access record from this master file address.
Step 3:	If this is an update, then update master index, master file, and transaction log.
(a) No differential file	
Step 1:	Search master index for record address.
Step 2:	Access record from either the master file or the differential file, depending on the address obtained in Step 1.
Step 3:	If this is an update, then update master index, differential file, and transaction log.
(b) Differential file in use	
Step 1:	Search differential index for record address. If the search is unsuccessful, then search the master index.
Step 2:	Access record from either the master file or the differential file, depending on the address obtained in Step 1.
Step 3:	If this is an update, then update differential index, differential file, and transaction log.
(c) Differential index and file in use	
Step 1:	Query the Bloom filter. If the answer is "maybe," then search differential index for record address. If the answer is "no" or if the differential index search is unsuccessful, then search the master index.
Step 2:	Access record from either the master file or the differential file, depending on the address obtained in Step 1.
Step 3:	If this is an update, then update Bloom filter, differential index, differential file, and transaction log.
(d) Differential index and file and Bloom filter in use	

Program 8.7: Access steps

8.4.2 Bloom Filter Design

The performance degradation that results from the use of a differential index can be considerably reduced by the use of a *Bloom filter*. This is a device that resides in internal memory and accepts queries of the following type: Is key k in the differential index? If queries of this type can be answered accurately, then there will never be a need to search both the differential and master indexes for a record address. Clearly, the only way to answer queries of this type accurately is to have a list of all keys in the differential index. This is not possible for differential indexes of reasonable size.

A Bloom filter does not answer queries of the above type accurately. Instead of returning one of "yes" and "no" as its answer, it returns one of "maybe" and "no". When the answer is "no," then we are assured that the key k is not in the differential index. In this case, only the master index is to be searched, and the number of disk accesses is the same as when a differential index is not used. If the answer is "maybe," then the differential index is searched. The master index needs to be searched only if k is not found in the differential index. Program 8.7(d) gives the steps to follow when a Bloom filter is used in conjunction with a differential index.

A *filter error* occurs whenever the answer to the Bloom filter query is "maybe" and the key is not in the differential index. Both the differential and master indexes are searched only when a filter error occurs. To obtain a performance close to that when a differential index is not in use, we must ensure that the probability of a filter error is close to zero.

Let us take a look at a Bloom filter. Typically, it consists of m bits of memory and k uniform and independent hash functions f_1, \dots, f_k . Each f_i hashes a key k to an integer in the range $[1, m]$. Initially all m filter bits are zero, and the differential index and file are empty. When key k is added to the differential index, bits $f_1(k), \dots, f_k(k)$ of the filter are set to 1. When a query of the type "Is key k in the differential index?" is made, bits $f_1(k), \dots, f_k(k)$ are examined. The query answer is "maybe" if all these bits are 1. Otherwise, the answer is "no." One may verify that whenever the answer is "no," the key cannot be in the differential index and that when the answer is "maybe," the key may or may not be in the differential index.

We can compute the probability of a filter error in the following way. Assume that initially there are n records and that n updates are made. Assume that none of these is an insert or a delete. Hence, the number of records remains unchanged. Further, assume that the record keys are uniformly distributed over the key space and that the probability that an update request is for record i is $1/n$, $1 \leq i \leq n$. From these assumptions, it follows that the probability that a particular update does not modify record i is $1 - 1/n$. So, the probability that none of the n updates modifies record i is $(1 - 1/n)^n$. Hence, the expected number of

unmodified records is $n(1 - 1/n)^h$, and the probability that the $(u+1)$ st update is for an unmodified record is $(1 - 1/n)^n$.

Next, consider bit i of the Bloom filter and the hash function f_j , $1 \leq j \leq h$. Let k be the key corresponding to one of the u updates. Since f_j is a uniform hash function, the probability that $f_j(k) \neq i$ is $1 - 1/n$. As the h hash functions are independent, the probability that $f_j(k) \neq i$ for all h hash functions is $(1 - 1/n)^h$. If this is the only update, the probability that bit i of the filter is zero is $(1 - 1/n)^h$. From the assumption on update requests, it follows that the probability that bit i is zero following the u updates is $(1 - 1/n)^{uh}$. From this, we conclude that if after u updates we make a query for an unmodified record, the probability of a filter error is $1 - (1 - 1/n)^{uh}$. The probability, $P(u)$, that an arbitrary query made after u updates results in a filter error is this quantity times the probability that the query is for an unmodified record. Hence,

$$P(u) = (1 - 1/n)^n (1 - (1 - 1/n)^{uh})$$

Using the approximation

$$(1 - 1/x)^x \approx e^{-1/x}$$

for large x , we obtain

$$P(u) \approx e^{-n/n} (1 - e^{-uh/n})$$

when n and m are large.

Suppose we wish to design a Bloom filter that minimizes the probability of a filter error. This probability is highest just before the master index is reorganized and the differential index becomes empty. Let u denote the number of updates done up to this time. In most applications, m is determined by the amount of memory available, and n is fixed. So, the only variable in design is h . Differentiating $P(u)$ with respect to h and setting the result to zero yields

$$h = (\log_e 2)m/n \approx 0.693m/n$$

We may verify that this h yields a minimum for $P(u)$. Actually, since h has to be an integer, the number of hash functions to use either is $\lceil 0.693m/n \rceil$ or $\lfloor 0.693m/n \rfloor$, depending on which one results in a smaller $P(u)$.

EXERCISES

1. By differentiating $P(u)$ with respect to h , show that $P(u)$ is minimized when $h = (\log_e 2)m/n$.
2. Suppose that you are to design a Bloom filter with minimum $P(u)$ and that $n = 100,000$, $m = 5000$, and $u = 1000$.
 - (a) Using any of the results obtained in the text, compute the number, h , of hash functions to use. Show your computations.

- (b) What is the probability, $P(u)$, of a filter error when h has this value?

8.5 REFERENCES AND SELECTED READINGS

For more on hashing, see *The Art of Computer Programming: Sorting and Searching*, by D. Knuth, Vol. 3, Second Edition, Addison-Wesley, Reading, MA, 1998 and "Hash tables", by P. Morin, in *Handbook of data structures and algorithms*, edited by D. Mehta and S. Sahni, Chapman & Hall/CRC, Boca Raton, 2005.

Our development of differential files and Bloom filters parallels that of Severance and Lohman in the paper "Differential files: Their application to the maintenance of large databases," by D. Severance and G. Lohman, *ACM Transactions on Database Systems*, 1:3, 1976, pp. 256-267. This paper also provides several advantages of using differential files. The assumptions of uniformity made in the filter error analysis are unrealistic, as, in practice, future accesses are more likely to be for records previously accessed. Several authors have attempted to take this into account. Two references are "A practical guide to the design of differential file architectures," by H. Aghili and D. Severance, *ACM Transactions on Database Systems*, 7:2, 1982, pp. 540-565; and "A regression approach to performance analysis for the differential file architecture," by T. Hill and A. Srinivasan, *Proceedings of the Third IEEE International Conference on Data Engineering*, 1987, pp. 157-164.

Bloom filters have found application in the solution to problems in a variety of domains. Some applications to network related problems may be found in "Space-code Bloom filter for efficient traffic flow measurement," by A. Kumar, J. Xu, L. Li and J. Wang, *ACM Internet Measurement Conference*, 2003; "Hash-based paging and location update using Bloom filters," by P. Mutaf and C. Castelluccia, *Mobile Networks and Applications*, Kluwer Academic, 9, 627-631, 2004; and "Approximate caches for packet classification," by F. Chang, W. Feng and K. Li, *IEEE INFOCOM*, 2004.

Priority Queues

9.1 SINGLE- AND DOUBLE-ENDED PRIORITY QUEUES

A *priority queue* is a collection of elements such that each element has an associated priority. We study two varieties of priority queues—single- and double-ended—in this chapter. Single-ended priority queues, which were first studied in Section 5.6, may be further categorized as min and max priority queues. As noted in Section 5.6.1, the operations supported by a min priority queue are:

SP1: Return an element with minimum priority.

SP2: Insert an element with an arbitrary priority.

SP3: Delete an element with minimum priority.

The operations supported by a max priority queue are the same as those supported by a min priority queue except that in SP1 and SP3 we replace minimum by maximum. The heap structure of Section 5.6 is a classic data structure for the representation of a priority queue. Using a min (max) heap, the

minimum (maximum) element can be found in $O(1)$ time and each of the other two single-ended priority queue operations may be done in $O(\log n)$ time, where n is the number of elements in the priority queue. In this chapter, we consider several extensions of a single-ended priority queue. The first extension, *meldable* (single-ended) priority queue, augments the operations SP1 through SP3 with a *meld* operation that melds together two priority queues. One application for the meld operation is when the server for one priority queue shuts down. At this time, it is necessary to meld its priority queue with that of a functioning server. Two data structures for meldable priority queues—leftist trees and binomial heaps—are developed in this chapter.

A further extension of meldable priority queues includes operations to delete an arbitrary element (given its location in the data structure) and to decrease the key/priority (or to increase the key, in case of a max priority queue) of an arbitrary element (given its location in the data structure). Two data structures—Fibonacci heaps and pairing heaps—are developed for this extension. The section on Fibonacci heaps describes how Fibonacci heaps may be used to improve the run time of Dijkstra's shortest paths algorithm of Section 6.4.1.

A *double-ended priority queue* (DEPQ) is a data structure that supports the following operations on a collection of elements.

- DP1: Return an element with minimum priority.
- DP2: Return an element with maximum priority.
- DP3: Insert an element with an arbitrary priority.
- DP4: Delete an element with minimum priority.
- DP5: Delete an element with maximum priority.

So, a DEPQ is a min and a max priority queue rolled into one structure.

Example 9.1: A DEPQ may be used to implement a network buffer. This buffer holds packets that are waiting their turn to be sent out over a network link; each packet has an associated priority. When the network link becomes available, a packet with the highest priority is transmitted. This corresponds to a *DeleteMax* operation. When a packet arrives at the buffer from elsewhere in the network, it is added to this buffer. This corresponds to an *insert* operation. However, if the buffer is full, we must drop a packet with minimum priority before we can insert one. This is achieved using a *DeleteMin* operation. □

Example 9.2: In Section 7.10, we saw how to adapt merge sort to the external sorting environment. We now consider a similar adaptation of quick sort, which has the best expected runtime of all known internal sorting methods. Recall that the basic idea in quick sort (Section 7.3) is to partition the elements to be sorted

into three groups L , M , and R . The middle group M contains a single element called the *pivot*, all elements in the left group L are \leq the pivot, and all elements in the right group R are \geq the pivot. Following this partitioning, the left and right element groups are sorted recursively.

In an external sort (Section 7.10), we have more elements than can be held in the memory of our computer. The elements to be sorted are initially on a disk and the sorted sequence is to be left on the disk. When the internal quick sort method outlined above is extended to an external quick sort, the middle group M is made as large as possible through the use of a DEPQ. The external quick sort strategy is:

- (1) Read in as many elements as will fit into an internal DEPQ. The elements in the DEPQ will eventually be the middle group of elements.
- (2) Process the remaining elements one at a time. If the next element is \leq the smallest element in the DEPQ, output this next element as part of the left group. If the next element is \geq the largest element in the DEPQ, output this next element as part of the right group. Otherwise, remove either the max or min element from the DEPQ (the choice may be made randomly or alternately); if the max element is removed, output it as part of the right group; otherwise, output the removed element as part of the left group; insert the newly input element into the DEPQ.
- (3) Output the elements in the DEPQ, in sorted order, as the middle group.
- (4) Sort the left and right groups recursively. \square

Program 9.1 defines an abstract base class *DEPQ* consisting of a pure virtual member function for each double-ended priority queue operation. There are many heap inspired data structures for the representation of a DEPQ. We develop just two of these—symmetric min-max heaps and interval heaps—in this chapter.

Throughout this chapter, we assume that priority queue elements are of a data type T that supports the relational operators ($=$, $<$, $>$, etc.) and that these operators compare element priorities. So, for example, one element is less than another iff it has a smaller priority.

```

template <class T>
class DEPQ {
public:
    virtual const T& GetMin() const = 0;
    virtual const T& GetMax() const = 0;
    virtual void Insert(const T&) = 0;
    virtual void DeleteMin() = 0;
    virtual void DeleteMax() = 0;
};

```

Program 9.1: Class definition of a double-ended priority queue

9.2 LEFTIST TREES

9.2.1 Height-Biased Leftist Trees

Leftist trees provide an efficient implementation of meldable priority queues. Consider the meld operation. Let n be the total number of elements in the two priority queues (throughout this section, we use the term *priority queue* to mean single-ended priority queue) that are to be melded. If heaps are used to represent meldable priority queues, then the meld operation takes $\tilde{O}(n)$ time (this may, for example, be accomplished using the heap initialization algorithm of Section 7.6). Using a leftist tree, the meld operation as well as the insert and delete min (or delete max) operations take logarithmic time; the minimum (or maximum) element may be found in $O(1)$ time.

Leftist trees are defined using the concept of an extended binary tree. An *extended binary tree* is a binary tree in which all empty binary subtrees have been replaced by a square node. Figure 9.1 shows two examples of binary trees. Their corresponding extended binary trees are shown in Figure 9.2. The square nodes in an extended binary tree are called *external nodes*. The original (circular) nodes of the binary tree are called *internal nodes*.

There are two varieties of leftist trees—height biased (HBLT) and weight biased (WBLT). We study HBLTs in this section and WBLTs in the next. HBLTs were invented first and are generally referred to simply as leftist trees. We continue with this tradition and refer to HBLTs simply as leftist trees in this section.

Let x be a node in an extended binary tree. Let $LeftChild(x)$ and $RightChild(x)$, respectively, denote the left and right children of the internal node x . Define $shortest(x)$ to be the length of a shortest path from x to an external node. It is easy to see that $shortest(x)$ satisfies the following recurrence:

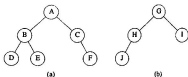


Figure 9.1: Two binary trees

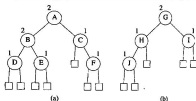


Figure 9.2: Extended binary trees corresponding to Figure 9.1

$$\text{shortest}(x) = \begin{cases} 0 & \text{if } x \text{ is an external node} \\ 1 + \min \{ \text{shortest}(\text{LeftChild}(x)), \text{shortest}(\text{RightChild}(x)) \} & \text{otherwise} \end{cases}$$

The number outside each internal node x of Figure 9.2 is the value of $\text{shortest}(x)$.

494 Priority Queues

Definition: A *leftist tree* is a binary tree such that if it is not empty, then

$$\text{shortest}(\text{LeftChild}(x)) \geq \text{shortest}(\text{RightChild}(x))$$

for every internal node x . \square

The binary tree of Figure 9.1(a), which corresponds to the extended binary tree of Figure 9.2(a), is not a leftist tree, as $\text{shortest}(\text{LeftChild}(C)) = 0$, whereas $\text{shortest}(\text{RightChild}(C)) = 1$. The binary tree of Figure 9.1(b) is a leftist tree.

Lemma 9.1: Let r be the root of a leftist tree that has n (internal) nodes.

- (a) $n \geq 2^{\text{shortest}(r)} - 1$
- (b) The rightmost root to external node path is the shortest root to external node path. Its length is $\text{shortest}(r) \leq \log_2(n+1)$.

Proof: (a) From the definition of $\text{shortest}(r)$ it follows that there are no external nodes on the first $\text{shortest}(r)$ levels of the leftist tree. Hence, the leftist tree has at least

$$\sum_{i=1}^{\text{shortest}(r)} 2^{i-1} = 2^{\text{shortest}(r)} - 1$$

internal nodes. (b) This follows directly from the definition of a leftist tree. \square

Leftist trees are represented using nodes that have the following data members: *leftChild*, *rightChild*, *shortest*, and *data*. We assume that the data type T of data overloads the relational operators ($<$, $>$, $=$, etc.) as comparisons between element priorities. We note that the concept of an external node is introduced merely to arrive at clean definitions. The external nodes are never physically present in the representation of a leftist tree. Rather the appropriate child data member (*leftChild* or *rightChild*) of the parent of an external node is set to NULL (or 0). Program 9.2 contains the class definitions of *LeftistNode* and *MinLeftistTree*.

Definition: A *min leftist tree* (or *leftist tree*) is a leftist tree in which the key value in each node is no larger (smaller) than the key values in its children (if any). In other words, a min (max) leftist tree is a leftist tree that is also a min (max) tree. \square

Two min leftist trees are shown in Figure 9.3. The number inside a node x is the priority of the element in x , and the number outside x is $\text{shortest}(x)$. For convenience, all priority queue figures in this chapter show only element priority

```
template <class T> class MinLeftistTree; // forward declaration
```

```
template <class T>
class LeftistNode {
friend class MinLeftistTree<T>;
private:
    T data;
    LeftistNode *leftChild, *rightChild;
    int shortest;
};

template <class T>
class MinLeftistTree : public MinPQ<T> {
public:
    // constructor
    MinLeftistTree(LeftistNode<T> *init = 0) root(init) { };

    // the four min leftist tree operations
    const T& GetMin() const;
    void Insert(const T&);
    T& DeleteMin();
    void Meld(MinLeftistTree<T>*);
private:
    LeftistNode<T>* Meld(LeftistNode<T>*, LeftistNode<T>*&);
    LeftistNode<T>* root;
};
```

Program 9.2: Class definition of a leftist tree

rather than the complete element. The operations insert, delete min (delete-max), and meld can be performed in logarithmic time using a min (max) leftist tree. We shall continue our discussion using min leftist trees.

The insert and delete-min operations can both be performed by using the meld operation. To insert an element x into a min leftist tree, we first create a min leftist tree that contains the single element x . Then we meld the two min leftist trees. To delete the min element from a nonempty min leftist tree, we meld the min leftist trees $root \rightarrow \text{leftChild}$ and $root \rightarrow \text{rightChild}$ and delete the node $root$.

The meld operation itself is quite simple. Suppose that two min leftist trees are to be melded. First, a new binary tree containing all elements in both trees is

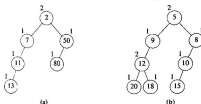


Figure 9.3: Examples of min leftist trees

obtained by following the rightmost paths in one or both trees. This binary tree has the property that the key in each node is no larger than the keys in its children (if any). Next, the left and right subtrees of nodes are interchanged as necessary to convert this binary tree into a leftist tree.

As an example, consider melding the two min leftist trees of Figure 9.3. To obtain a binary tree that contains all the elements in each tree and that satisfies the required relationship between parent and child keys, we first compare the root keys 2 and 5. Since $2 < 5$, the new binary tree should have 2 in its root. We shall leave the left subtree of 2 unchanged and meld 2's right subtree with the entire binary tree rooted at 5. The resulting binary tree will become the new right subtree of 2. When melding the right subtree of 2 and the binary tree rooted at 5, we notice that $5 < 50$. So, 5 should be in the root of the melded tree. Now, we proceed to meld the subtrees with root 8 and 50. Since $8 < 50$ and 8 has no right subtree, we can make the subtree with root 50 the right subtree of 8. This gives us the binary tree of Figure 9.4(a). Hence, the result of melding the right subtree of 2 and the tree rooted at 5 is the tree of Figure 9.4(b). When the tree of Figure 9.4(b) is made the right subtree of 2, we get the binary tree of Figure 9.4(c). The leftist tree that is made a subtree is represented by shading its nodes. In each step. To convert the tree of Figure 9.4(c) into a leftist tree, we begin at the last modified root (i.e., 8) and trace back to the overall root, ensuring that $\text{shorten}(\text{LeftChild}(i)) \geq \text{shorten}(\text{RightChild}(i))$. This inequality holds at 8 but not at 5 and 2. Simply interchanging the left and right subtrees at these nodes causes the inequality to hold. The result is the leftist tree of Figure 9.4(d). The

pointers that were interchanged are represented by dotted lines in the figure.

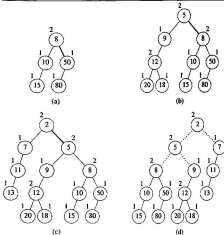


Figure 9.4: Merging the min leftist trees of Figure 9.3

The function to meld two leftist trees is given in Program 9.3. This function makes use of the recursive function *Meld*(*a*, *b*), which melds two nonempty leftist trees. This recursive function intertwines the following two steps:

- (1) create a min tree that contains all the elements
- (2) ensure that each node has a left subtree whose shorter value is greater than

```

template <class T>
void MinLeftistTree<T>::Meld(MinLeftistTree<T> *b)
// Combine the min leftist tree b with the given min leftist tree.
// b is set to the empty min leftist tree.
    if (!root) root = b->root;
    else if (b->root) root = Meld(root, b->root);
    b->root = 0;
}

template <class T>
LeftistNode<T>* MinLeftistTree<T>::Meld(LeftistNode<T> *a,
                                         LeftistNode<T> *b)
// Recursive function to meld two nonempty min leftist trees rooted
// at a and b. The root of the resulting min leftist tree is returned.
// Set a to be min leftist tree with smaller root.
    if (a->data > b->data) swap(a, b);

    // create binary tree such that the smallest key in each subtree is in the root
    if (a->rightChild) a->rightChild = b;
    else a->rightChild = Meld(a->rightChild, b);

    // leftist tree property
    if ((a->rightChild && a->leftChild->shortest < a->rightChild->shortest)
        // interchange subtrees
        swap(a->leftChild, a->rightChild);

    // set shortest data member
    if (!a->rightChild) a->shortest = 1;
    else a->shortest = a->rightChild->shortest + 1;
    return a;
}

```

Program 9.3: Melding two min leftist trees

or equal to that of its right subtree

Analysis of $Meld()$ ($LeftistNode<T>*$, $LeftistNode<T>*$): Since $Meld$ moves down the rightmost paths in the two leftist trees being melded, and since the lengths of these paths are at most logarithmic in the number of elements in each tree (Lemma 9.1), the melding of two leftist trees with a total of n elements is done in time $O(\log n)$. \square

9.2.2 Weight-Biased Leftist Trees

We arrive at another variety of leftist tree by considering the number of nodes in a subtree, rather than the length of a shortest root to external node path. Define the weight $w(x)$ of node x to be the number of internal nodes in the subtree with root x . Notice that if x is an external node, its weight is 0. If x is an internal node, its weight is 1 more than the sum of the weights of its children. The weights of the nodes of the binary trees of Figure 9.2 appear in Figure 9.5.

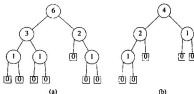


Figure 9.5: Extended binary trees of Figure 9.2 with weights shown

Definition: A binary tree is a *weight-biased leftist tree (WBLT)* iff at every internal node the w value of the left child is greater than or equal to the w value of the right child. A max (min) WBLT is a max (min) tree that is also a WBLT. \square

Note that the binary tree of Figure 9.5(a) is not a WBLT while that of Figure 9.5(b) is.

Lemma 9.2: Let x be any internal node of a weight-biased leftist tree. The length, $rightmost(x)$, of the rightmost path from x to an external node satisfies

$$rightmost(x) \leq \log_2(w(x)+1).$$

Proof: The proof is by induction on $w(x)$. When $w(x)=1$, $rightmost(x)=1$ and $\log_2(w(x)+1)=\log_2 2=1$. For the induction hypothesis, assume that $rightmost(x) \leq \log_2(w(x)+1)$ whenever $w(x) < n$. Let $RightChild(x)$ denote the

500 Priority Queues

right child of x (note that this right child may be an external node). When $w(x)=n$, $w(\text{RightChild}(x)) \leq (n-1)/2$ and

$$\begin{aligned} \text{rightmost}(x) &= 1 + \text{rightmost}(\text{RightChild}(x)) \\ &\leq 1 + \log_2((n-1)/2 + 1) \\ &= 1 + \log_2(n+1) - 1 \\ &= \log_2(n+1). \quad \square \end{aligned}$$

The insert, delete max, and initialization operations are analogous to the corresponding max HBLT operations. However, the meld operation can be done in a single top-to-bottom pass (recall that the meld operation of an HBLT performs a top-to-bottom pass as the recursion unfolds and then a bottom-to-top pass in which subtrees are possibly swapped and *shortest*-values updated). A single-pass meld is possible for WBLTs because we can determine the w -values on the way down and so, on the way down, we can update w -values and swap subtrees as necessary. For HBLTs, a node's new *shortest* value cannot be determined on the way down the tree.

Experiments indicate that meldable single-ended priority queue operations are faster, by a constant factor, when we use WBLTs rather than HBLTs.

EXERCISES

1. Give an example of a binary tree that is not a leftist tree. Label the nodes of your binary tree with their *shortest* value.
2. Let t be an arbitrary binary tree represented using the node structure for a leftist tree.
 - (a) Write a function to initialize the *shortest* data member of each node in t .
 - (b) Write a function to convert t into a leftist tree.
 - (c) What is the complexity of each of these two functions?
3.
 - (a) Into an empty min leftist tree, insert elements with priorities 20, 10, 5, 18, 6, 12, 14, 4, and 22 (in this order). Show the min leftist tree following each insert.
 - (b) Delete the min element from the final min leftist tree of part (a). Show the resulting min leftist tree.
4. Compare the performance of leftist trees and min heaps under the assumption that the only operations to be performed are insert and delete min. For this, do the following:
 - (a) Create a random list of n elements and a random sequence of insert and delete-min operations of length m . The latter sequence is created such that the probability of an insert or delete-min operation

is approximately 0.5. Initialize a min leftist tree and a min heap to contain the n elements in the first random list. Now, measure the time to perform the m operations using the min leftist tree as well as the min heap. Divide this time by m to get the average time per operation. Do this for $n = 100, 500, 1000, 2000, \dots, 5000$. Let m be 5000. Tabulate your computing times.

- (b) Based on your experiments, make some statements about the relative merits of the two priority-queue schemes.
5. Write a function to initialize a min leftist tree with n elements. Assume that the node structure is the same as that used in the text. Your function must run in $\Theta(n)$ time. Show that this is the case. Can you think of a way to do this initialization in $\Theta(n)$ time such that the resulting min leftist tree is also a complete binary tree?
6. Fully code and test the class *MinLeftistTree* of Program 9.2.
7. Write a function to delete the element in node x of a min leftist tree. Assume that each node has the data members *leftChild*, *rightChild*, *parent*, *shortest*, and *data*. The *parent* data member of a node points to its parent in the leftist tree. What is the complexity of your function?
8. [*Lazy deletion*] Another way to handle the deletion of arbitrary elements from a min leftist tree is to use a *bool* data member, *deleted*, in place of the *parent* data member of the previous exercise. When an element is deleted, its *deleted* data member is set to *true*. However, the node is not physically deleted. When a delete-min operation is performed, we first search for the minimum element not deleted by performing a limited preorder search. This preorder search traverses only the upper part of the tree as needed to identify the min element. All deleted elements encountered are physically deleted, and their subtrees are melded to obtain the new min leftist tree.
 - (a) Write a function to delete the element in node x of a min leftist tree.
 - (b) Write another function to delete the min element from a min leftist tree from which several elements have been deleted using the former function.
 - (c) What is the complexity of your function of part (b)? Provide this as a function of the number of deleted elements encountered and the number of elements in the entire tree?
9. [*Skew heaps*] A *skew heap* is a min tree that supports the min leftist tree operations: insert, delete min, and meld in amortized time (see Section 9.4 for a definition of amortized time) $O(\log n)$ per operation. As in the case of min leftist trees, insertions and deletions are performed using the meld operation, which is carried out by following the rightmost paths in the two heaps being melded. However, unlike min leftist trees, the left and right

502 Priority Queues

subtrees of all nodes (except the last) on the rightmost path in the resulting heap are interchanged.

- (a) Write insert, delete min, and meld functions for skewed heaps.
 - (b) Compare the running times of these with those for the same operations on a min leftist tree. Use random sequences of insert, delete min, and meld operations.
10. [WBLT] Develop the class *MixWblt* that implements a weight-biased min leftist tree. You must include functions to delete and return the min element, insert an arbitrary element, and meld two min WBLTs. Your meld function should perform only a top-to-bottom pass over the WBLTs being melded. Show that the complexity of these three functions is $O(n)$. Test all functions using your own test data.
11. Give an example of an HBLT that is not a WBLT as well as one that is a WBLT but not an HBLT.

9.3 BINOMIAL HEAPS

9.3.1 Cost Amortization

A *binomial heap* is a data structure that supports the same functions (i.e., insert, delete min (or delete-max), and meld) as those supported by leftist trees. Unlike leftist trees, where an individual operation can be performed in $O(\log n)$ time, it is possible that certain individual operations performed on a binomial heap may take $O(n)$ time. However, if we amortize (spread out) part of the cost of expensive operations over the inexpensive ones, then the amortized complexity of an individual operation is either $O(1)$ or $O(\log n)$ depending on the type of operation.

Let us examine more closely the concept of cost amortization (we shall use the terms *cost* and *complexity* interchangeably). Suppose that a sequence $I1, I2, D1, I3, I4, I5, I6, D2, I7$ of insert and delete-min operations is performed. Assume that the actual cost of each of the seven inserts is one. By this, we mean that each insert takes one unit of time. Further, suppose that the delete-min operations $D1$ and $D2$ have an actual cost of eight and ten, respectively. So, the total cost of the sequence of operations is 25.

In an amortization scheme we charge some of the actual cost of an operation to other operations. This reduces the charged cost of some operations and increases that of others. The *amortized cost* of an operation is the total cost charged to it. The cost transferring (amortization) scheme is required to be such that the sum of the amortized costs of the operations is greater than or equal to the sum of their actual costs. If we charge one unit of the cost of a delete-min

operation to each of the inserts since the last delete-min operation (if any), then two units of the cost of D1 get transferred to I1 and I2 (the charged cost of each increases by one), and four units of the cost of D2 get transferred to I3 to I6. The amortized cost of each of I1 to I6 becomes two, that of I7 is equal to its actual cost (i.e., one), and that of each of D1 and D2 becomes 6. The sum of the amortized costs is 25, which is the same as the sum of the actual costs.

Now suppose we can prove that no matter what sequence of insert and delete-min operations is performed, we can charge costs in such a way that the amortized cost of each insertion is no more than two and that of each deletion is no more than six. This will enable us to make the claim that the actual cost of any insert / delete min sequence is no more than $2i + 6d$ where i and d are, respectively, the number of insert and delete min operations in the sequence. Suppose that the actual cost of a deletion is no more than ten, and that of an insertion is one. Using actual costs, we can conclude that the sequence cost is no more than $i + 10d$. Combining these two bounds, we obtain $\min(2i + 6d, i + 10d)$ as a bound on the sequence cost. Hence, using the notion of cost amortization, we can obtain tighter bounds on the complexity of a sequence of operations. This is useful, because in many applications, we are concerned more with the time it takes to perform a sequence of priority queue operations than we are with the time it takes to perform an individual operation. For example, when we sort using the heap-sort method, we are concerned with the time it takes to complete the entire sort; not with the time it takes to remove the next element from the heap. In applications such as sorting, where we are concerned only with the overall time rather than the time per operation, it is adequate to use a data structure that has a good amortized complexity for each operation type.

We shall use the notion of cost amortization to show that although individual delete operations on a binomial heap may be expensive, the cost of any sequence of binomial heap operations is actually quite small.

9.3.2 Definition of Binomial Heaps

As in the case of heaps and leftist trees, there are two varieties of binomial heaps, min and max. A *min binomial heap* is a collection of min trees; a *max binomial heap* is a collection of max trees. We shall explicitly consider min binomial heaps only. These will be referred to as *B-heaps*. Figure 9.6 shows an example of a B-heap that is made up of three min trees.

Using B-heaps, we can perform an insert and a meld in $O(1)$ actual and amortized time and a delete min in $O(\log n)$ amortized time. B-heaps are represented using nodes that have the following data members: *degree*, *child*, *link*, and *data*. The *degree* of a node is the number of children it has; the *child*



Figure 9.6: A B-heap with three min trees

data member is used to point to any one of its children (if any); the *link* data member is used to maintain singly linked circular lists of siblings. All the children of a node form a singly linked circular list, and the node points to one of these children. Additionally, the roots of the min trees that comprise a B-heap are linked to form a singly linked circular list. The B-heap is then pointed at by a single pointer *min* to the min tree root with smallest key. Program 9.4 contains the class definitions for *BinomialNode* and *BinomialHeap*.

Figure 9.7 shows the representation for the example of Figure 9.6. To enhance the readability of this figure, we have used bidirectional arrows to join together nodes that are in the same circular list. When such a list contains only one node, no such arrows are drawn. Each of the key sets {10}, {6}, {5, 4}, {30}, {15, 30}, {9}, {12, 7, 16}, and {8, 3, 1} denotes the keys in one of the circular lists of Figure 9.7. *min* is the pointer to the B-heap. Note that an empty B-heap has a 0 pointer.

9.3.3 Insertion into a Binomial Heap

An element x may be inserted into a B-heap by first putting x into a new node and then inserting this node into the circular list pointed at by *min*. The pointer *min* is reset to this new node only if *min* is 0 or the key of x is smaller than the key in the node pointed at by *min*. It is evident that these insertion steps can be performed in $O(1)$ time.

```
template <class T> class BinomialHeap; // forward declaration
```

```
template <class T>
class BinomialNode {
friend class BinomialHeap<T>;
private:
    T data;
    BinomialNode<T> *child, *link;
    int degree;
};

template <class T>
class BinomialHeap : public MinPQ<T> {
public:
    BinomialHeap(BinomialNode<T> *init = 0) min(init) { };

    // the four binomial heap operations
    const T& GetMin() const;
    void Insert(const T&);
    T& DeleteMin();
    void Merge(BinomialHeap<T>*&);

private:
    BinomialNode<T> *root;
};
```

Program 9.4: Class definition of a binomial heap

9.3.4 Merging Two Binomial Heaps

To meld two nonempty B-heaps, we meld the top circular lists of each into a single circular list. The new B-heap pointer is the *min* pointer of one of the two trees, depending on which has the smaller key. This can be determined with a single comparison. Since two circular lists can be melded into a single one in $O(1)$ time, the total time required to meld two B-heaps is $O(1)$.

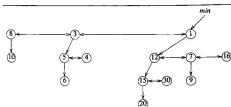


Figure 9.7: B-heap of Figure 9.6 showing child pointers and sibling lists

9.3.5 Deletion of Min Element

If *min* is 0, then the B-heap is empty, and a deletion cannot be performed. Assume that *min* is not 0. *min* points to the node that contains the min element. This node is deleted from its circular list. The new B-heap consists of the remaining min trees and the sub-min trees of the deleted root. Figure 9.8 shows the situation for the example of Figure 9.6.



Figure 9.8: The B-heap of Figure 9.6 following the deletion of the min element

Before forming the circular list of min tree roots, we repeatedly join

together pairs of min trees that have the same degree (the degree of a nonempty min tree is the degree of its root). This min tree joining is done by making the min tree whose root has a larger key a subtree of the other (ties are broken arbitrarily). When two min trees are joined, the degree of the resulting min tree is one larger than the original degree of each min tree, and the number of min trees decreases by one. For our example, we may first join either the min trees with roots 8 and 7 or those with roots 3 and 12. If the first pair is joined, the min tree with root 8 is made a subtree of the min tree with root 7. We now have the min tree collection of Figure 9.9. There are three min trees of degree two in this collection. If the pair with roots 7 and 3 is picked for joining, the resulting min tree collection is that of Figure 9.10. Shaded nodes in Figure 9.9 and Figure 9.10 denote the min tree that was made a subtree in the previous step. Since the min trees in this collection have different degrees, the min tree joining process terminates.



Figure 9.9: The B-heap of Figure 9.8 following the joining of the two degree-one min trees

The min tree joining step is followed by a step in which the min tree roots are linked together to form a circular list and the B-heap pointer is reset to point to the min tree root with smallest key. The steps involved in a delete-min operation are summarized in Program 9.5.

Step 1 takes $O(1)$ time. Step 3 can be done in $O(1)$ time by copying over the data from the next node (if any) in the circular list and physically deleting that node instead. However, since Step 3 requires us to visit all nodes in the circular list of roots, it is not necessary to delete *min* and leave behind a circular list. In Step 3, we can simply examine all nodes other than *min* in the root-level circular list.

Step 3 may be implemented by using an array, *tree*, that is indexed from 0 to the maximum possible degree, *maxDegree*, of a min tree. Initially all entries in this array are 0. Let *s* be the number of min trees in *min* and *y*. The lists *min*

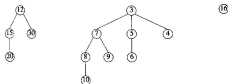


Figure 9.10: The B-heap of Figure 9.9 following the joining of two degree-two min trees

Step 1: [Handle empty B-heap] if (!min) throw QueueEmpty();

Step 2: [Deletion from nonempty B-heap] $x = \text{min} \rightarrow \text{data}$; $y = \text{min} \rightarrow \text{child}$; delete min from its circular list; following this deletion, min points to any remaining node in the resulting list; if there is no such node, then $\text{min} = 0$;

Step 3: [Min-tree joining] Consider the min trees in the lists min and y. Join together pairs of min trees of the same degree until all remaining min trees have different degrees;

Step 4: [Form min tree root list] Link the roots of the remaining min trees (if any) together to form a circular list; set min to point to the root (if any) with minimum key; return x;

Program 9.5: Steps in a delete-min operation

and y created in Step 2 are scanned. For each min tree p in the lists min and y created in Step 2, the code of Program 9.6 is executed. The function *JoinMin-Trees* makes the input tree with larger root a subtree of the other tree. The resulting tree is returned in the first parameter. In the end, the array *tree* contains pointers to the min trees that are to be linked together in Step 4. Since each time

a pair of min trees is joined the total number of min trees decreases by one, the number of joins is at most $r-1$. Hence, the complexity of Step 3 is $O(\text{maxDegree} + r)$.

```

for ( $d = p \rightarrow \text{degree}$ ;  $\text{tree}[d]$ ;  $d++$ )
{
    JoinMinTrees( $p$ ,  $\text{tree}[d]$ );
     $\text{tree}[d] = 0$ ;
}
 $\text{tree}[d] = p$ ;

```

Program 9.6: Code to handle min tree p encountered during a scan of lists x and y

Step 4 is accomplished by scanning $\text{tree}[0], \dots, \text{tree}[\text{maxDegree}]$ and linking together the min trees found. During this scan, the min tree with minimum key may also be determined. The complexity of Step 4 is $O(\text{maxDegree})$.

9.3.6 Analysis

Definition: The *binomial tree*, B_k , of degree k is a tree such that if $k = 0$, then the tree has exactly one node, and if $k > 0$, then the tree consists of a root whose degree is k and whose subtrees are B_0, B_1, \dots, B_{k-1} . \square

The min trees of Figure 9.6 are B_1, B_2 , and B_3 , respectively. One may verify that B_k has exactly 2^k nodes. Further, if we start with a collection of empty B-heaps and perform inserts, melds, and delete-mins only, then the min trees in each B-heap are binomial trees. These observations enable us to prove that when only inserts, melds, and delete-mins are performed, we can amortize costs such that the amortized cost of each insert and meld is $O(1)$, and that of each delete-min is $O(\log n)$.

Lemma 9.3: Let α be a B-heap with n elements that results from a sequence of insert, meld, and delete-min operations performed on a collection of initially empty B-heaps. Each min tree in α has degree $\leq \log_2 n$. Consequently, $\text{maxDegree} \leq \lceil \log_2 n \rceil$, and the actual cost of a delete-min operation is $O(\log n + r)$.

Proof: Since each of the min trees in α is a binomial tree with at most n nodes, none can have degree greater than $\lceil \log_2 n \rceil$. \square

510 Priority Queues

Theorem 9.1: If a sequence of n insert, meld, and delete-min operations is performed on initially empty B-heaps, then we can amortize costs such that the amortized time complexity of each insert and meld is $O(1)$, and that of each delete-min operation is $O(\log n)$.

Proof: For each B-heap, define the quantities $\#insert$ and $LastSize$ in the following way: When an initially empty B-heap is created or when a delete-min operation is performed on a B-heap, its $\#insert$ value is set to zero. Each time an insert is done on a B-heap, its $\#insert$ value is increased by one. When two B-heaps are melded, the $\#insert$ value of the resulting B-heap is the sum of the $\#insert$ values of the B-heaps melded. Hence, $\#insert$ counts the number of inserts performed on a B-heap or its constituent B-heaps since the last delete-min operation performed in each. When an initially empty B-heap is created, its $LastSize$ value is zero. When a delete-min operation is performed on a B-heap, its $LastSize$ is set to the number of min trees it contains following this delete. When two B-heaps are melded, the $LastSize$ value for the resulting B-heap is the sum of the $LastSize$ values in the two B-heaps that were melded. One may verify that the number of min trees in a B-heap is always equal to $\#insert + LastSize$.

Consider any individual delete-min operation in the operation sequence. Assume this is from the B-heap a . Observe that the total number of elements in all the B-heaps is at most n , as only inserts add elements, and at most n inserts can be present in a sequence of n operations. Let $a = a_i$, $\min\text{-degree} \leq \log_2 n$.

From Lemma 9.3, the actual cost of this delete-min operation is $O(\log n + c)$. The $\log n$ term is due to \maxDegree and represents the time needed to initialize the array *tree* and to complete Step 4. The c term represents the time to scan the lists *min* and *y* and to perform the $x-1$ (at most) min tree joins. We see that $x = \#insert + LastSize + a - 1$. If we charge $\#insert$ units of cost to the insert operations that contribute to the count $\#insert$ and $LastSize$ units to the delete mins that contribute to the count $LastSize$ (each such delete-min operation is charged a number of cost units equal to the number of min trees it left behind), then only $a - 1$ of the x cost units remains. Since $a \leq \log_2 n$, and since the number of min trees in a B-heap immediately following a delete-min operation is $\leq \log_2 n$, the amortized cost of a delete-min operation becomes $O(\log_2 n)$.

Since this charging scheme adds at most one unit to the cost of any insert, the amortized cost of an insert becomes $O(1)$. The amortization scheme used does not charge anything extra to a meld. So, the actual and amortized costs of a meld are also $O(1)$. \square

From the preceding theorem and the definition of cost amortization, it follows that the actual cost of any sequence of i inserts, c melds, and dm delete-min operations is $O(i + c + dm \log n)$.

EXERCISES

1. Let S be an initially empty stack. We wish to perform two operations on S : *Add*(x) and *DeleteUntil*(x). These are defined as follows:
 - (a) *Add*(x): Add the element x to the top of the stack S . This operation takes $O(1)$ time per invocation.
 - (b) *DeleteUntil*(x): Delete elements from the top of the stack up to and including the first x encountered. If p elements are deleted, the time taken is $O(p)$.

Consider any sequence of n stack operations (*Add* and *DeleteUntil*). Show how to amortize the cost of the *Add* and *DeleteUntil* operations so that the amortized cost of each is $O(1)$. From this, conclude that the time needed to perform any such sequence of operations is $O(n)$.

2. Let x be an unsorted array of n elements. The function *Search*(x, n, i, y) searches x for y by examining $x[i]$, $x[i+1]$, and so on, in that order, for the least j such that $x[j] = y$. If no such j is found, j is set to $n+1$. On termination, function *Search* sets i to j . Assume that the time required to examine a single element of x is $O(1)$.
 - (a) What is the worst-case complexity of *Search*?
 - (b) Suppose that a sequence of m searches is performed beginning with $i = 1$. Use a cost amortization scheme that assigns costs both to elements and to search operations. Show that it is always possible to amortize costs so that the amortized cost of each element is $O(1)$ and that of each search is also $O(1)$. From this, conclude that the cost of the sequence of m searches is $O(m + n)$.
3. (a) Into an empty B-heap, insert elements with priorities 20, 10, 5, 18, 6, 12, 14, 4, and 22 (in this order). Show the final B-heap.
 - (b) Delete the min element from the final B-heap of part (a). Show the resulting B-heap. Show how you arrived at this final B-heap.
4. Fully code and test the class *BHeap*, which implements a min binomial heap. Your class must include the functions *GetMin*, *Insert*, *DeleteMin*, and *Meld*.
5. Prove that the binomial tree B_k has 2^k nodes, $k \geq 0$.
6. Compare the performance of leftist trees and B-heaps under the assumption that the only permissible operations are insert and delete min. For this, do the following:
 - (a) Create a random list of n elements and a random sequence of insert and delete-min operations of length m . The number of delete mins and inserts should be approximately equal. Initialize a min leftist

512 Priority Queues

tree and a B-heap to contain the n elements in the first random list. Now, measure the time to perform the n operations using the min leftist tree as well as the B-heap. Divide this time by n to get the average time per operation. Do this for $n = 100, 500, 1000, 2000, \dots, 5000$. Let n be 5000. Tabulate your computing times.

- (b) Based on your experiments, make some statements about the relative merits of the two data structures.
7. Is the height of every tree in a Binomial heap that has n elements $O(\log n)$? If not, what is the worst-case height as a function of n ?

9.4 FIBONACCI HEAPS

9.4.1 Definition

There are two varieties of Fibonacci heaps: min and max. A *min Fibonacci heap* is a collection of min trees; a *max Fibonacci heap* is a collection of max trees. We shall explicitly consider min Fibonacci heaps only. These will be referred to as *F-heaps*. B-heaps are a special case of F-heaps. Thus, all the examples of B-heaps in the preceding section are also examples of F-heaps. As a consequence, in this section, we shall refer to these examples as F-heaps.

An *F-heap* is a data structure that supports the four B-heap operations—*GetMin*, *Insert*, *DeleteMin*, and *Meld*—as well as the following additional operations:

- (1) *Delete*: Delete the element in a specified node. We refer to this delete operation as *arbitrary delete*.
- (2) *DecreaseKey*: Decrease the key/priority of a specified node by a given positive amount.

When an F-heap is used, the *Delete* operation takes $O(\log n)$ amortized time and the *DecreaseKey* takes $O(1)$ amortized time. The B-heap operations can be performed in the same asymptotic times using an F-heap as they can be using a B-heap.

To represent an F-heap, the B-heap representation is augmented by adding two data members, *parent* and *childCur*, to each node. The *parent* data member is used to point to the node's parent (if any). The significance of the *childCur* data member will be described later. In addition, the singly linked circular lists are replaced by doubly linked circular lists. This requires us to replace the data member *link* by the data members *leftLink* and *rightLink*.

In an F-heap, the B-heap operations *GetMin*, *Insert*, *DeleteMin*, and *Meld* are performed exactly as for the case of B-heaps. So, we need consider only the

remaining two operations: *Delete* and *DecreaseKey*.

9.4.2 Deletion from an F-Heap

To delete an arbitrary node b from a F-heap, we do the following:

- (1) If $\text{min} = b$, then do a delete min; otherwise do Steps 2, 3, and 4 below.
- (2) Delete b from its doubly linked list.
- (3) Combine the doubly linked list of b 's children with the doubly linked list pointed at by min into a single doubly linked list. Trees of equal degree are not joined as in a delete-min operation.
- (4) Dispose of node b .

For example, if we delete the node containing 12 from the F-heap of Figure 9.6, we get the F-heap of Figure 9.11. The actual cost of an arbitrary delete is $O(1)$ unless the min element is being deleted. In this case the deletion time is the time for a delete-min operation.



Figure 9.11: F-heap of Figure 9.6 following the deletion of 12

9.4.3 Decrease Key

To decrease the key in node b we do the following:

- (1) Reduce the key in b .
- (2) If b is not a min tree root and its key is smaller than that in its parent, then

514 Priority Queues

delete b from its doubly linked list and insert it into the doubly linked list of min tree roots.

- (3) Change min to point to b if the key in b is smaller than that in min .

Suppose we decrease the key 15 in the F-heap of Figure 9.6 by 4. The resulting F-heap is shown in Figure 9.12. The cost of performing a decrease-key operation is $O(1)$.



Figure 9.12: F-heap of Figure 9.6 following the reduction of 15 by 4

9.4.4 Cascading Cut

With the addition of the delete and decrease-key operations, the min trees in an F-heap need not be binomial trees. In fact, it is possible to have degree- k min trees with as few as $k+1$ nodes. As a result, the analysis of Theorem 9.1 is no longer valid. The analysis of Theorem 9.1 requires that each min tree of degree k have an exponential (in k) number of nodes. When decrease-key and delete operations are performed as described above, this is no longer true. To ensure that each min tree of degree k has at least c^k nodes for some c , $c > 1$, each delete and decrease-key operation must be followed by a *cascading-cut* step. For this, we add the bool data member *childCut* to each node. The value of this data member is useful only for nodes that are not the root of a min tree. In this case, the *childCut* data member of node x has the value true iff one of the children of x was cut off (i.e., removed) after the most recent time x was made the child of its current parent. This means that each time two min trees are joined in a delete-min operation, the *childCut* data member of the root with larger key should be set to false. Further, whenever a delete or decrease-key operation deletes a node q that is not a min tree root from its doubly linked list (Step 2 of delete and

decrease key), then the cascading-cut step is invoked. During this step, we examine the nodes on the path from the parent p of the deleted node q up to the nearest ancestor of the deleted node with $childCut = \text{false}$. If there is no such ancestor, then the path goes from p to the root of the min tree containing p . All nonroot nodes on this path with $childCut$ data member true are deleted from their respective doubly linked lists and added to the doubly linked list of min tree root nodes of the F-heap. If the path has a node with $childCut$ data member false, this data member is changed to true.

Figure 9.13 gives an example of a cascading cut. Figure 9.13(a) is the min tree containing 14 before a decrease-key operation that reduces this key by 4. The $childCut$ data members are shown only for the nodes on the path from the parent of 14 to its nearest ancestor with $childCut = \text{false}$. Nodes with $childCut = \text{true}$ are shaded in the figure. All unshaded nodes have $childCut = \text{false}$. During the decrease-key operation, the min tree with root 14 is deleted from the min tree of Figure 9.13(a) and becomes a min tree of the F-heap. Its root now has key 10. This is the first min tree of Figure 9.13(b). During the cascading cut, the min trees with roots 12, 10, 8, and 6 are cut off from the min tree with root 2. Thus, the single min tree of Figure 9.13(a) becomes six min trees of the resulting F-heap. The $childCut$ value of 4 becomes true. All other $childCut$ values are unchanged.

9.4.5 Analysis

Lemma 9.4: Let a be an F-heap with n elements that results from a sequence of insert, meld, delete min, delete, and decrease-key operations performed on initially empty F-heaps.

- (a) Let b be any node in any of the min trees of a . The degree of b is at most $\log_4 n$, where $\phi = (1 + \sqrt{5})/2$, and n is the number of elements in the subtree with root b .
- (b) $\text{maxDegree} \leq \lceil \log_4 n \rceil$, and the actual cost of a delete-min operation is $O(\log n + 1)$.

Proof: We shall prove (a) by induction on the degree of b . Let N_i be the minimum number of elements in the subtree with root b when b has degree i . We see that $N_0 = 1$ and $N_1 = 2$. So, the inequality of (a) holds for degrees 0 and 1. For $i > 1$, let c_1, \dots, c_i be the i children of b . Assume that c_j was made a child of b before c_{j+1} , $j < i$. Hence, when c_k , $k \leq i$, was made a child of b , the degree of b was at least $k - 1$. The only F-heap operation that makes one node a child of another is delete min. Here, during a join min tree step, one min tree is made a subtree of another min tree of equal degree. Hence, at the time of joining, the

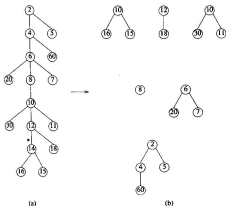


Figure 9.13: A cascading cut following a decrease of key 14 by 4

degree of c_k must have been equal to that of b . Subsequent to joining, its degree can decrease as a result of a delete or decrease-key operation. However, following such a join, the degree of c_k can decrease by at most one, as an attempt to cut off a second child of c_k results in a cascading cut at c_k . Such a cut causes c_k to become the root of a min tree of the P-heap. Hence, the degree, d_k , of c_k is at least $\max(0, k-2)$. So, the number of elements in c_k is at least N_k . This implies that

$$N_1 = N_0 + \sum_{k=0}^{i-2} N_k + 1 = \sum_{k=0}^{i-2} N_k + 2$$

One may show (see the exercises) that the Fibonacci numbers satisfy the equality

$$F_h = \sum_{k=0}^{h-2} F_k + 1, \quad h > 1, \quad F_0 = 0, \text{ and } F_1 = 1$$

From this we may obtain the equality $N_i = F_{i+2}$, $i \geq 0$. Further, since $F_{i+2} \geq \psi^i$, $N_i \geq \psi^i$. Hence, $i \leq \log_\psi m$. (b) is a direct consequence of (a). \square

Theorem 9.2: If a sequence of n insert, meld, delete min, delete, and decrease-key operations is performed on an initially empty F-heap, then we can amortize costs such that the amortized time complexity of each insert, meld, and decrease-key operation is $O(1)$ and that of each delete min and delete operation is $O(\log n)$. The total time complexity of the entire sequence is the sum of the amortized complexities of the individual operations in the sequence.

Proof: The proof is similar to that of Theorem 9.1. The definition of *#insert* is unchanged. However, that of *LastSize* is augmented by requiring that following each delete and decrease-key operation, *LastSize* be changed by the net change in the number of min trees in the F-heap (in the example of Figure 9.13 *LastSize* is increased by 5). With this modification, we see that at the time of a delete-min operation $s = \#insert + LastSize + s - 1$. *#insert* units of cost may be charged, one each, to the *#insert* insert operations that contribute to this count, and *LastSize* units may be charged to the delete min, delete, and decrease-key operations that contribute to this count. This results in an additional charge of at most $\log_\psi n$ to each contributing delete min and delete operation and of one to each contributing decrease-key operation. As a result, the amortized cost of a delete-min operation is $O(\log n)$.

Since the total number of cascading cuts is limited by the total number of deletes and decrease-key operations (as these are the only operations that can set *ChildCut* to true), the cost of these cuts may be amortized over the delete and decrease-key operations by adding one to their amortized costs. The amortized cost of deleting an element other than the min element becomes $O(\log n)$, as its actual cost is $O(1)$ (excluding the cost of the cascading-cut sequence that may be performed); at most one unit is charged to it from the amortization of all the cascading cuts; and at most $\log_\psi n$ units are charged to it from a delete-min operation.

The amortized cost of a decrease-key operation is $O(1)$, as its actual cost is $O(1)$ (excluding the cost of the ensuing cascading cut); at most one unit is charged to it from the amortization of all cascading cuts; and at most one unit is charged from a delete-min operation.

The amortized cost of an insert is $O(1)$, as its actual cost is one, and at most one cost unit is charged to it from a delete-min operation. Since the amortization scheme transfers no charge to a meld operation, its actual and amortized costs

5.18 Priority Queues

are the same. This cost is $O(1)$. \square

From the preceding theorem, it follows that the complexity of any sequence of F-heap operations is $O(i + c + dk + (dm + d) \log i)$ where i , c , dk , dm , and d are, respectively, the number of insert, meld, decrease-key, delete min, and delete operations in the sequence.

9.4.6 Application to the Shortest-Paths Problem

We conclude this section on F-heaps by considering their application to the single-source/all-destinations algorithm of Chapter 6. Let S be the set of vertices to which a shortest path has been found and let $\text{dist}(i)$ be the length of a shortest path from the source vertex to vertex i , $i \in S$, that goes through only vertices in S . On each iteration of the shortest-path algorithm, we need to determine an i , $i \notin S$, such that $\text{dist}(i)$ is minimum and add this i to S . This corresponds to a delete-min operation on S . Further, the dist values of the remaining vertices in S may decrease. This corresponds to a decrease-key operation on each of the affected vertices. The total number of decrease-key operations is bounded by the number of edges in the graph, and the number of delete-min operations is $n - 2$. S begins with $n - 1$ vertices. If we implement S as an F-heap using dist as the key, then $n - 1$ inserts are needed to initialize the F-heap. Additionally, $n - 2$ delete-min operations and at most e decrease-key operations are needed. The total time for all these operations is the sum of the amortized costs for each. This is $O(n \log n + e)$. The remainder of the algorithm takes $O(e)$ time. Hence if an F-heap is used to represent S , the complexity of the shortest-path algorithm becomes $O(n \log n + e)$. This is an asymptotic improvement over the implementation discussed in Chapter 6 if the graph does not have $\Omega(n^2)$ edges. If this single-source algorithm is used n times, once with each of the n vertices in the graph as the source, then we can find a shortest path between every pair of vertices in $O(n^2 \log n + ne)$ time. Once again, this represents an asymptotic improvement over the $O(n^3)$ dynamic programming algorithm of Chapter 6 for graphs that do not have $\Omega(n^2)$ edges. It is interesting to note that $O(n \log n + e)$ is the best possible implementation of the single-source algorithm of Chapter 6, as the algorithm must examine each edge and may be used to sort n numbers (which takes $O(n \log n)$ time).

EXERCISES

1. Fully code and test the class *FHeap*, which implements a min Fibonacci heap. Your class must include the functions *GetMin*, *Insert*, *DeleteMin*, *Meld*, *Delete*, and *DecreaseKey*. The function *Insert* should return the node

into which the new element was inserted. This returned information can later be used as an input to *Delete* and *DecreaseKey*.

2. Prove that if we start with empty F-heaps and perform only the operations insert, meld, and delete min, then all min trees in the F-heaps are binomial trees.
3. Can all the functions on an F-heap be performed in the same amount of time using singly linked circular lists rather than doubly linked circular lists? (Note that we can delete from an arbitrary node x of a singly linked circular list by copying over the data in the next node and then deleting the next node rather than the node x .)
4. Show that if we start with empty F-heaps and do not perform cascading cuts, then it is possible for a sequence of F-heap operations to result in degree- k min trees that have only $k+1$ nodes, $k \geq 1$.
5. Is the height of every tree in a Fibonacci heap that has n elements $O(\log n)$? If not, what is the worst-case height as a function of n ?
6. Suppose we change the rule for a cascading cut so that such a cut is performed only when a node loses a third child rather than when it loses a second child. For this, the `childCnt` data member is changed so that it can have the values 0, 1, and 2. When a node acquires a new parent, its `childCnt` data member is set to 1. Each time a node has a child cut off (during a delete or decrease-key operation), its `childCnt` data member is increased by one (unless this data member is already two). If the `childCnt` data member is already two, a cascading cut is performed.
 - (a) Obtain a recurrence equation for N_i , the minimum number of nodes in a min tree with degree i . Assume that we start with an empty F-heap and that all operations (except cascading cut) are performed as described in the text. Cascading cuts are performed as described above.
 - (b) Solve the recurrence of part (a) to obtain a lower bound on N_i .
 - (c) Does the modified rule for cascading cuts ensure that the minimum number of nodes in any min tree of degree i is exponential in i ?
 - (d) For the new cascading-cut rule, can you establish the same amortized complexities as for the original rule? Prove the correctness of your answer.
 - (e) Answer parts (c) and (d) under the assumption that cascading cuts are performed only after k children of a node have been cut off. Here, k is a fixed constant ($k = 2$ for the rule used in the text, and $k = 3$ for the rule used earlier in this exercise).
 - (f) How do you expect the performance of F-heaps to change as larger

values of k (see part (e)) are used?

7. For the Fibonacci numbers F_k and the numbers N_i of Lemma 9.4, prove the following:

$$(a) \quad F_k = \sum_{i=0}^{k-2} F_i + 1, k \geq 1.$$

- (b) Use (a) to show that $N_i = F_{i+2}$, $i \geq 0$.

- (c) Use the equality

$$F_k = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^k - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^k, k \geq 0$$

to show that $F_{k+2} \geq \phi^k$, $k \geq 0$, where $\phi = (1+\sqrt{5})/2$.

8. Implement the single-source shortest-path algorithm of Chapter 6 using the data structures recommended there as well as P-heaps. However, use adjacency lists rather than an adjacency matrix. Generate 10 connected undirected graphs with different edge densities (say 10%, 20%, ..., 100% of maximum) for each of the cases $n = 100, 200, \dots, 500$. Assign random costs to the edges (use a uniform random number generator in the range [1, 1000]). Measure the run times of the two implementations of the shortest-path algorithm. Plot the average times for each n .

9.5 PAIRING HEAPS

9.5.1 Definition

The pairing heap supports the same operations as supported by the Fibonacci heap. Pairing heaps come in two varieties—min pairing heaps and max pairing heaps. Min pairing heaps are used when we wish to represent a min priority queue, and max pairing heaps are used for max priority queues. In keeping with our discussion of Fibonacci heaps, we explicitly discuss min pairing heaps only. Max pairing heaps are analogous. Figure 9.14 compares the actual and amortized complexities of the Fibonacci and pairing heap operations.

Although the amortized complexities given in Figure 9.14 for pairing heap operations are not known to be tight (i.e., no one knows of an operation sequence whose run time actually grows logarithmically with the number of decrease key operations (say)), it is known that the amortized complexity of the decrease key operation is $\Omega(\log \log n)$ (see the section titled References and Selected Readings at the end of this chapter).

Although the amortized complexity is better when a Fibonacci heap is used rather than when a pairing heap is used, extensive experimental studies employing these structures in the implementation of Dijkstra's shortest paths algorithm

Operation	Fibonacci Heap		Pairing Heap	
	Actual	Amortized	Actual	Amortized
<i>GetMin</i>	$O(1)$	$O(1)$	$O(1)$	$O(1)$
<i>Insert</i>	$O(1)$	$O(1)$	$O(1)$	$O(1)$
<i>DeleteMin</i>	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$
<i>Merge</i>	$O(1)$	$O(1)$	$O(1)$	$O(\log n)$
<i>Delete</i>	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$
<i>DecreaseKey</i>	$O(n)$	$O(1)$	$O(1)$	$O(\log n)$

Figure 9.14: Complexity of Fibonacci and pairing heap operations

(Section 6.4.1) and Prim's minimum cost spanning tree algorithm (Section 6.3.2) indicate that pairing heaps actually outperform Fibonacci heaps.

Definition: A *min pairing heap* is a min tree in which the operations are performed in a manner to be specified later.

Figure 9.15 shows four example min pairing heaps. Notice that a pairing heap is a single tree, which need not be a binary tree. The min element is in the root of this tree and hence this element may be found in $O(1)$ time.



Figure 9.15: Example min pairing heaps

9.5.2 Meld and Insert

Two min pairing heaps may be melded into a single min pairing heap by performing a *compare-link* operation. In a *compare-link*, the roots of the two min trees are compared and the min tree that has the larger root is made the leftmost subtree of the other tree (ties are broken arbitrarily).

To meld the min trees of Figures 9.15 (a) and (b), we compare the two roots. Since tree (a) has the larger root, this tree becomes the leftmost subtree of tree (b). Figure 9.16 (a) is the resulting pairing heap. Figure 9.16 (b) shows the result of melding the pairing heaps of Figures 9.15 (c) and (d). When we meld the pairing heaps of Figures 9.16 (a) and (b), the result is the pairing heap of Figure 9.17.



Figure 9.16: Melding pairing heaps

To insert an element x into a pairing heap p , we first create a pairing heap q with the single element x , and then meld the two pairing heaps p and q .

9.5.3 Decrease Key

Suppose we decrease the key/priority of the element in node N . When N is the root or when the new key in N is greater than or equal to that in its parent, no additional work is to be done. However, when the new key in N is less than that in its parent, the min tree property is violated and corrective action is to be taken. For example, if the key in the root of the tree of Figure 9.15 (c) is decreased from 1 to 0, or when the key in the leftmost child of the root of Figure 9.15 (c) is decreased from 4 to 2 no additional work is necessary. When the key in the leftmost child of the root of Figure 9.15 (c) is decreased from 4 to 0 the new value is



Figure 9.17: Meld of Figures 9.16 (a) and (b)

less than that in the root (see Figure 9.18 (a)) and corrective action is needed.

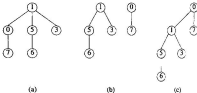


Figure 9.18: Decreasing a key

Since pairing heaps are normally not implemented with a parent pointer, it is difficult to determine whether or not corrective action is needed following a key reduction. Therefore, corrective action is taken regardless of whether or not it is needed except when N is the tree root. The corrective action consists of the following steps:

524 Priority Queues

Step 1: Remove the subtree with root N from the tree. This results in two min trees.

Step 2: Meld the two min trees together.

Figure 9.18 (b) shows the two min trees following Step 1, and Figure 9.18 (c) shows the result following Step 2.

9.5.4 Delete Min

The min element is in the root of the tree. So, to delete the min element, we first delete the root node. When the root is deleted, we are left with zero or more min trees (i.e., the subtrees of the deleted root). When the number of remaining min trees is two or more, these min trees must be melded into a single min tree. In *two pass pairing heaps*, this melding is done as follows:

Step 1: Make a left to right pass over the trees, melding pairs of trees.

Step 2: Start with the rightmost tree and meld the remaining trees (right to left) into this tree one at a time.

Consider the min pairing heap of Figure 9.19 (a). When the root is removed, we get the collection of 6 min trees shown in Figure 9.19 (b).



Figure 9.19: Deleting the min element

In the left to right pass of Step 1, we first meld the trees with roots 4 and 0. Next, the trees with roots 3 and 5 are melded. Finally, the trees with roots 1 and 6 are melded. Figure 9.20 shows the resulting three min trees.



Figure 9.20: Trees following first pass

In Step 2 (which is a right to left pass), the two rightmost trees of Figure 9.20 are first melded to get the tree of Figure 9.21 (a).



Figure 9.21: First stage of second pass

Then the tree of Figure 9.20 with root 0 is melded with the tree of Figure 9.21 to get the final min tree, which is shown in Figure 9.22.

Note that if the original pairing heap had 8 subtrees, then following the left to right melding pass we would be left with 4 min trees. In the right to left pass, we would first meld trees 3 and 4 to get tree 5. Then trees 2 and 5 would be melded to get tree 6. Finally, we would meld trees 1 and 6.

In *multi pass pairing heaps*, the min trees that remain following the removal of the root are melded into a single min tree as follows:

Step 1: Put the min trees onto a FIFO queue.

Step 2: Extract two trees from the front of the queue, meld them and put the

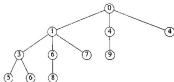


Figure 9.22: Final min pairing heap following a delete min

resulting tree at the end of the queue. Repeat this step until only one tree remains.

Consider the six trees of Figure 9.19 (b) that result when the root of Figure 9.19 (a) is deleted. First, we meld the trees with roots 4 and 0 and put the resulting min tree at the end of the queue. Next, the trees with roots 3 and 5 are melded and the resulting min tree is put at the end of the queue. And then, the trees with roots 1 and 6 are melded and the resulting min tree added to the queue end. The queue now contains the three min trees shown in Figure 9.20. Next, the min trees with roots 0 and 3 are melded and the result put at the end of the queue. We are now left with the two min trees shown in Figure 9.23.

Finally, the two min trees of Figure 9.23 are melded to get the min tree of Figure 9.24.

9.5.5 Arbitrary Delete

Deletion from an arbitrary node N is handled as a delete-min operation when N is the root of the pairing heap. When N is not the tree root, the deletion is done as follows:

Step 1: Detach the subtree with root N from the tree.

Step 2: Delete node N and meld its subtrees into a single min tree using the two pass scheme if we are implementing a two pass pairing heap or the multi



Figure 9.23: Next to last state in multi pass delete

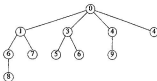


Figure 9.24: Result of multi pass delete min

pass scheme if we are implementing a multi pass pairing heap.

Step 3: Mold the min trees from Steps 1 and 2 into a single min tree.

9.5.6 Implementation Considerations

Although we can implement a pairing heap using nodes that have a variable number of children fields, such an implementation is expensive because of the need to dynamically increase the number of children fields as needed. An

528 Priority Queues

efficient implementation results when we use the binary tree representation of a tree (see Section 3.1.2.2). Siblings in the original min tree are linked together using a doubly linked list. In addition to a *data* field, each node has the three pointer fields *previous*, *next*, and *child*. The leftmost node in a doubly linked list of siblings uses its *previous* pointer to point to its parent. A leftmost child satisfies the property $x \rightarrow \text{previous} \rightarrow \text{child} = x$. The doubly linked list makes it possible to remove an arbitrary element (as is required by the *Delete* and *DecreaseKey* operations) in $O(1)$ time.

9.5.7 Complexity

You can verify that using the described binary tree representation, all pairing heap operations (other than *Delete* and *DeleteMin*) can be done in $O(1)$ time. The complexity of the *Delete* and *DeleteMin* operations is $O(n)$, because the number of subtrees that have to be melded following the removal of a node is $O(n)$.

The amortized complexity of the pairing heap operations is established in the paper by Fredman et al. cited in the References and Selected Readings section. Experimental studies conducted by Stasko and Vitter (see their paper that is cited in the References and Selected Readings section) establish the superiority of two pass pairing heaps over multipass pairing heaps.

EXERCISES

- Into an empty two pass min pairing heap, insert elements with priorities 20, 10, 5, 18, 6, 12, 14, 9, 8 and 22 (in this order). Show the min pairing heap following each insert.
 - Delete the min element from the final min pairing heap of part (a). Show the resulting pairing heap.
- Into an empty multi pass min pairing heap, insert elements with priorities 20, 10, 5, 18, 6, 12, 14, 9, 8 and 22 (in this order). Show the min pairing heap following each insert.
 - Delete the min element from the final min pairing heap of part (a). Show the resulting pairing heap.
- Fully code and test the class *MultiPassPairingHeaps*, which implements a multi pass min pairing heap. Your class must include the functions *GetMin*, *Insert*, *DeleteMin*, *Meld*, *Delete* and *DecreaseKey*. The function *Insert* should return the node into which the new element was inserted. This returned information can later be used as an input to *Delete* and *DecreaseKey*.

4. What are the worst-case height and degree of a pairing heap that has n elements? Show how you arrived at your answer.
5. Define a *one pass pairing heap* as an adaptation of a two pass pairing heap in which Step 1 (Make a left to right pass over the trees, melding pairs of trees.) is eliminated. Show that the amortized cost of either insert or delete min must be $\Theta(n)$.

9.6 SYMMETRIC MIN-MAX HEAPS

9.6.1 Definition and Properties

A double-ended priority queue (DEPQ) may be represented using a symmetric min-max heap (SMMH). An *SMMH* is a complete binary tree in which each node other than the root has exactly one element. The root of an SMMH is empty and the total number of nodes in the SMMH is $n+1$, where n is the number of elements. Let N be any node of the SMMH. Let *elements*(N) be the elements in the subtree rooted at N but excluding the element (if any) in N . Assume that *elements*(N) $\neq \emptyset$. N satisfies the following properties:

Q1: The left child of N has the minimum element in *elements*(N).

Q2: The right child of N (if any) has the maximum element in *elements*(N).

Figure 9.25 shows an example SMMH that has 12 elements. When N denotes the node with 80, *elements*(N) = {6, 14, 30, 40}: the left child of N has the minimum element 6 in *elements*(N); and the right child of N has the maximum element 40 in *elements*(N). You may verify that every node N of this SMMH satisfies properties Q1 and Q2.

It is easy to see that an $n+1$ -node complete binary tree with an empty root and one element in every other node is an SMMH iff the following are true:

P1: The element in each node is less than or equal to that in its right sibling (if any).

P2: For every node N that has a grandparent, the element in the left child of the grandparent is less than or equal to that in N .

P3: For every node N that has a grandparent, the element in the right child of the grandparent is greater than or equal to that in N .

Properties P2 and P3, respectively, state that the grandchildren of each node M have elements that are greater than or equal to that in the left child of M and less than or equal to that in the right child of M . Hence, P2 and P3 follow from Q1 and Q2, respectively. Notice that if property P1 is satisfied, then at most one of P2 and P3 may be violated at any node N . Using properties P1



Figure 9.25: A symmetric min-max heap

through P3 we arrive at simple algorithms to insert and delete elements. These algorithms are simple adaptations of the corresponding algorithms for heaps.

As we shall see, the standard DEPQ operations of Program 9.1 can be done efficiently using an SMMH.

9.6.2 SMMH Representation

Since an SMMH is a complete binary tree, it is efficiently represented as a one-dimensional array (say A) using the standard mapping of a complete binary tree into an array (Section 5.2.3.1). Position 0 of A is not used and position 1, which represents the root of the complete binary tree, is empty. We use the variable *last* to denote the rightmost position of A in which we have stored an element of the SMMH. So, the size (i.e., number of elements) of the SMMH is *last* - 1. The variable *arrayLength* keeps track of the current number of positions in the array A .

When $n=1$, the minimum and maximum elements are the same and are in the left child of the root of the SMMH. When $n>1$, the minimum element is in the left child of the root and the maximum is in the right child of the root. So, the *GetMin* and *GetMax* operations take $O(1)$ time each. Program 9.7 defines the class *SMMH*, which implements a symmetric min-max heap. Program 9.8 gives the constructor that creates an empty SMMH.

```

template <class T>
class SMMH : public DEPQ {
public:
    SMMH(int initialCapacity = 10);
    ~SMMH() {delete [] h;}

    const T& GetMin() const
    { // return min element
      if (last == 1) throw QueueEmpty();
      return h[2];
    }

    const T& GetMax() const
    { // return max element
      if (last == 1) throw QueueEmpty();
      if (last == 2) return h[2];
      else return h[3];
    }

    void Insert(const T&);
    void DeleteMin();
    void DeleteMax();
private:
    int last;           // position of last element in queue
    int arrayLength;    // queue capacity + 2
    T *h;               // element array
};

```

Program 9.7: Class definition for symmetric min-max heap

9.6.3 Inserting into an SMMH

The algorithm to insert into an SMMH has three steps.

- Step 1:** Expand the size of the complete binary tree by 1, creating a new node E for the element x that is to be inserted. This newly created node of the complete binary tree becomes the candidate node to insert the new element x .
- Step 2:** Verify whether the insertion of x into E would result in a violation of property P1. Note that this violation occurs iff E is a right child of its

```

template <class T>
SMMH<T>::SMMH(int initialCapacity)
// Constructor.
    if (initialCapacity < 1)
    {
        ostream s;
        s << "Initial capacity = " << initialCapacity << " Must be >= 0";
        throw IllegalParameterValue(s, m());
    }
    arrayLength = initialCapacity + 2;
    h = new T[arrayLength];
    last = 1;
}

```

Program 9.8 Constructor for SMMH

parent and x is greater than the element in the sibling of E . In case of a P1 violation, the element in the sibling of E is moved to E and E is updated to be the new empty sibling.

Step 3: Perform a bubble-up pass from E up the tree verifying properties P2 and P3. In each round of the bubble-up pass, E moves up the tree by one level. When E is positioned so that the insertion of x into E doesn't result in a violation of either P2 or P3, insert x into E .

Suppose we wish to insert 2 into the SMMH of Figure 9.25. Since an SMMH is a complete binary tree, we must add a new node to the SMMH in the position shown in Figure 9.26; the new node is labeled E . In our example, E will denote an empty node.

If the new element 2 is placed in node E , property P2 is violated as the left child of the grandparent of E has 6. So we move the 6 down to E and move E up one level to obtain the configuration of Figure 9.27.

Now we determine if it is safe to insert the 2 into node E . We first notice that such an insertion cannot result in a violation of property P1, because the previous occupant of node E was greater than 2. For properties P2 and P3, let $N=E$. P3 cannot be violated for this value of N as the previous occupant of this node was greater than 2. So, only P2 can be violated. Checking P2 with $N=E$, we see that P2 will be violated if we insert $x=2$ into E , because the left child of the grandparent of E has the element 4. So we move the 4 down to E and move E up one level to the node that previously contained the 4. Figure 9.28 shows the resulting configuration.

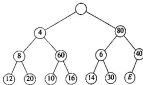


Figure 9.26: The SMMH of Figure 9.25 with a node added

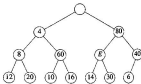


Figure 9.27: The SMMH of Figure 9.26 with 6 moved down

For the configuration of Figure 9.28 we see that placing 2 into node E cannot violate property P1, because the previous occupant of node E was greater than 2. Also properties P2 and P3 cannot be violated, because node E has no grandparent. So we insert 2 into node E and obtain Figure 9.29.

Let us now insert 50 into the SMMH of Figure 9.29. Since an SMMH is a complete binary tree, the new node must be positioned as in Figure 9.30.

Since E is the right child of its parent, we first check P1 at node E . If the

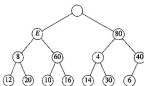


Figure 9.28: The SMMH of Figure 9.27 with 4 moved down

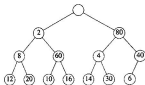


Figure 9.29: The SMMH of Figure 9.28 with 2 inserted

new element (in this case 30) is smaller than that in the left sibling of E , we swap the new element and the element in the left sibling. In our case, no swap is done. Then we check $P2$ and $P3$. We see that placing 30 into E would violate $P3$. So the element 40 in the right child of the grandparent of E is moved down to node E . Figure 9.31 shows the resulting configuration. Placing 30 into node E of Figure 9.31 cannot create a $P1$ violation because the previous occupant of node E was smaller. A $P2$ violation isn't possible either. So only $P3$ needs to be checked

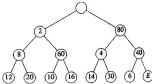


Figure 9.30: The SMMH of Figure 9.29 with a node added

at E . Since there is no P3 violation at E , 50 is placed into E .

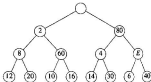


Figure 9.31: The SMMH of Figure 9.30 with 40 moved down

Program 9.9 gives the C++ code for the insert operation; the variable `currentNode` refers to the empty node E of our example. Since the height of a complete binary tree is $O(\log n)$ and Program 9.9 does $O(1)$ work at each level of the SMMH, the complexity of the insert function is $O(\log n)$.

```

template <class T>
void SMMH<T>::Insert(const T& x)
{ // Insert x into the SMMH.
  // increase array length if necessary
  if (last == arrayLength - 1)
  { // double array length
    ChangeSize(D, arrayLength, 2 * arrayLength);
    arrayLength *= 2;
  }
  // find place for x
  // currentNode starts at new leaf and moves up tree
  int currentNode = ++last;
  if (last % 2 == 1 && x < h[last - 1])
  { // left sibling must be smaller, P1
    h[last] = h[last - 1];
    currentNode--;
  }
  bool done = false;
  while (!done && currentNode >= 4)
  { // currentNode has a grandparent
    int gp = currentNode / 4; // grandparent
    int lggp = 2 * gp; // left child of gp
    int rggp = lggp + 1; // right child of gp
    if (x < h[lggp])
    { // P2 is violated
      h[currentNode] = h[lggp];
      currentNode = lggp;
    }
    else if (x > h[rggp])
    { // P3 is violated
      h[currentNode] = h[rggp];
      currentNode = rggp;
    }
    else done = true; // neither P2 nor P3 violated
  }
  h[currentNode] = x;
}

```

Program 9.9: Insertion into a symmetric min-max heap

9.6.4 Deleting from an SMMH

The algorithm to delete either the min or max element is an adaptation of the trickle-down algorithm used to delete an element from a min or a max heap. We consider only the case when the minimum element is to be deleted. If the SMMH is empty, the deletion cannot be performed. So, assume we have a non-empty SMMH. The minimum element is in $h[2]$. If $last=2$, the SMMH becomes empty following the deletion. Assume that $last \neq 2$. Let $x=h[last]$ and decrement $last$ by 1. To complete the deletion, we must reinsert x into an SMMH whose $h[2]$ node is empty. Let E denote the empty node. We follow a path from E down the tree, as in the delete algorithm for a min or max heap, verifying properties P1 and P2 until we reach a suitable node into which x may be inserted. In the case of a delete-min operation, the trickle-down process cannot cause a P3 violation. So, we don't explicitly verify P3.

Consider the SMMH of Figure 9.31 with 50 in the node labeled E . A delete min results in the removal of 2 from the left child of the root (i.e., $h[2]$) and the removal of the last node (i.e., the one with 40) from the SMMH. So, $x=40$ and we have the configuration shown in Figure 9.32. Since $h[3]$ has the maximum element, P1 cannot be violated at E . Further, since E is the left child of its parent, no P3 violations can result from inserting x into E . So we need only concern ourselves with P2 violations. To detect such a violation, we determine the smaller of the left child of E and the left child of E 's right sibling. For our example, the smaller of 8 and 4 is determined. This smaller element 4 is, by definition of an SMMH, the smallest element in the SMMH. Since $4 < x=40$, inserting x into E would result in a P2 violation. To avoid this, we move the 4 into node E and the node previously occupied by 4 becomes E (see Figure 9.33). Notice that if $4 \geq x$, inserting x into E would result in a properly structured SMMH.

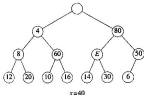
Now the new E becomes the candidate node for the insertion of x . First, we check for a possible P1 violation that may result from such an insertion. Since $x=40 < 50$, no P1 violation results. Then we check for a P2 violation. The left children of E and its sibling are 14 and 6. The smaller child, 6, is smaller than x . So, x cannot be inserted into E . Rather, we swap E and 6 to get the configuration of Figure 9.34.

We now check the P1 property at the new E . Since E doesn't have a right sibling, there is no P1 violation. We proceed to check the P2 property. Since E has no children, a P2 violation isn't possible either. So, x is inserted into E . Let's consider another delete-min operation. This time, we delete the minimum element from the SMMH of Figure 9.34 (recall that the node labeled E contains 40). The min element 4 is removed from $h[2]$ and the last element, 40, is removed from the SMMH and placed in x . Figure 9.35 shows the resulting configuration.

As before, a P1 violation isn't possible at $h[2]$. The smaller of the left



Figure 9.32: The SMMH of Figure 9.31 with 2 deleted

Figure 9.33: The SMMH of Figure 9.32 with E and 4 interchanged

children of E and its sibling is 6. Since $6 < x=40$, we interchange 6 and E to get Figure 9.36.

Next, we check for a possible P1 violation at the new E . Since the sibling of E is 50 and $x=40 \leq 50$, no P1 violation is detected. The smaller left child of E

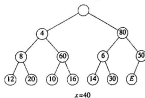
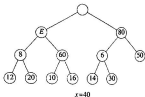

 Figure 9.34: The 3MHH of Figure 9.33 with E and 6 interchanged


Figure 9.35: First step of another delete min

and its sibling is 14 (actually, the sibling doesn't have a left child, so we just use the left child of E), which is $< x$. So, we swap E and 14 to get Figure 9.37.

Since there is a P1 violation at the new E , we swap x and 30 to get Figure 9.38 and proceed to check for a P2 violation at the new E . As there is no P2 violation here, $x=30$ is inserted into E .



Figure 9.36: The SMMH of Figure 9.35 with 8 and 6 interchanged



Figure 9.37: The SMMH of Figure 9.36 with 8 and 14 interchanged

We leave the development of the code for the delete operations as an exercise. However, you should note that these operations spend $O(1)$ time per level during the trickle-down pass. So, their complexity is $O(\log n)$.



Figure 9.38: The SMMH of Figure 9.37 with x and 30 interchanged

EXERCISES

1. Show that every complete binary tree with an empty root and one element in every other node is an SMMH iff P1 through P3 are true.
2. Start with an empty SMMH and insert the elements 20, 10, 40, 3, 2, 7, 60, 1 and 80 (in this order) using the insertion algorithm developed in this section. Draw the SMMH following each insert.
3. Perform 3 delete-min operations on the SMMH of Figure 9.38 with 30 in the node E . Use the delete min strategy described in this section. Draw the SMMH following each delete min.
4. Perform 4 delete max operations on the SMMH of Figure 9.38 with 30 in the node E . Adapt the delete min strategy of this section to the delete max operation. Draw the SMMH following each delete max operation.
5. Develop the code for all functions of the class *SMMH* (Program 9.7). Test all functions using your own test data.

9.7 INTERVAL HEAPS

9.7.1 Definition and Properties

Like an SMMH, an interval heap is a heap inspired data structure that may be used to represent a DEPQ. An *interval heap* is a complete binary tree in which each node, except possibly the last one (the nodes of the complete binary tree are ordered using a level order traversal), contains two elements. Let the two

542 Priority Queues

elements in a node be a and b , where $a \leq b$. We say that the node represents the closed interval $[a, b]$. a is the left end point of the node's interval and b is its right end point.

The interval $[c, d]$ is contained in the interval $[a, b]$ iff $a \leq c \leq d \leq b$. In an interval heap, the intervals represented by the left and right children (if they exist) of each node P are contained in the interval represented by P . When the last node contains a single element e , then $a \leq e \leq b$, where $[a, b]$ is the interval of the parent (if any) of the last node.

Figure 9.39 shows an interval heap with 16 elements. You may verify that the intervals represented by the children of any node P are contained in the interval of P .



Figure 9.39: An interval heap

The following facts are immediate:

- (1) The left end points of the node intervals define a min heap, and the right end points define a max heap. In case the number of elements is odd, the last node has a single element which may be regarded as a member of either the min or max heap. Figure 9.40 shows the min and max heaps defined by the interval heap of Figure 9.39.
- (2) When the root has two elements, the left end point of the root is the minimum element in the interval heap and the right end point is the maximum. When the root has only one element, the interval heap contains just one element. This element is both the minimum and maximum element.
- (3) An interval heap can be represented compactly by mapping into an array as

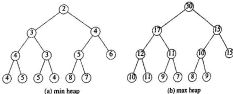


Figure 9.40: Min and max heaps embedded in Figure 9.39

is done for ordinary heaps. However, now, each array position must have space for two elements.

- (4) The height of an interval heap with n elements is $O(\log n)$.

9.7.2 Inserting into an Interval Heap

Suppose we are to insert an element into the interval heap of Figure 9.39. Since this heap currently has an even number of elements, the heap following the insertion will have an additional node A as is shown in Figure 9.41.

The interval for the parent of the new node A is $[6, 15]$. Therefore, if the new element is between 6 and 15, the new element may be inserted into node A . When the new element is less than the left end point 6 of the parent interval, the new element is inserted into the min heap embedded in the interval heap. This insertion is done using the min heap insertion procedure starting at node A . When the new element is greater than the right end point 15 of the parent interval, the new element is inserted into the max heap embedded in the interval heap. This insertion is done using the max heap insertion procedure starting at node A .

If we are to insert the element 10 into the interval heap of Figure 9.39, this element is put into the node A shown in Figure 9.41. To insert the element 3, we follow a path from node A towards the root, moving left end points down until we either pass the root or reach a node whose left end point is ≤ 3 . The new



Figure 9.41: Interval heap of Figure 9.39 after one node is added

element is inserted into the node that now has no left end point. Figure 9.42 shows the resulting interval heap.



Figure 9.42: The interval heap of Figure 9.39 with 3 inserted

To insert the element 40 into the interval heap of Figure 9.39, we follow a path from node A (see Figure 9.41) towards the root, moving right end points down until we either pass the root or reach a node whose right end point is ≥ 40 .

The new element is inserted into the node that now has no right end point. Figure 9.43 shows the resulting interval heap.



Figure 9.43: The interval heap of Figure 9.39 with 40 inserted

Now, suppose we wish to insert an element into the interval heap of Figure 9.43. Since this interval heap has an odd number of elements, the insertion of the new element does not increase the number of nodes. The insertion procedure is the same as for the case when we initially have an even number of elements. Let A denote the last node in the heap. If the new element lies within the interval $[6, 15]$ of the parent of A , then the new element is inserted into node A (the new element becomes the left end point of A if it is less than the element currently in A). If the new element is less than the left end point 6 of the parent of A , then the new element is inserted into the embedded min heap; otherwise, the new element is inserted into the embedded max heap. Figure 9.44 shows the result of inserting the element 32 into the interval heap of Figure 9.43.

9.7.3 Deleting the Min Element

The removal of the minimum element is handled as several cases:

- (1) When the interval heap is empty, the *DeleteMin* operation fails.
- (2) When the interval heap has only one element, this element is the element to be returned. We leave behind an empty interval heap.



Figure 9.44: The interval heap of Figure 9.43 with 32 inserted

- (3) When there is more than one element, the left end point of the root is to be retained. This point is removed from the root. If the root is the last node of the interval heap, nothing more is to be done. When the last node is not the root node, we remove the left point p from the last node. If this causes the last node to become empty, the last node is no longer part of the heap. The point p removed from the last node is reinserted into the embedded min heap by beginning at the root. As we move down, it may be necessary to swap the current p with the right end point r of the node being examined to ensure that $p \leq r$. The reinsertion is done using the same strategy as used to reinsert into an ordinary heap.

Let us remove the minimum element from the interval heap of Figure 9.44. First, the element 2 is removed from the root. Next, the left end point 15 is removed from the last node and we begin the reinsertion procedure at the root. The smaller of the min heap elements that are the children of the root is 3. Since this element is smaller than 15, we move the 3 into the root (the 3 becomes the left end point of the root) and position ourselves at the left child B of the root. Since, $15 \leq 17$ we do not swap the right end point of B with the current $p=15$. The smaller of the left end points of the children of B is 3. The 3 is moved from node C into node B as its left end point and we position ourselves at node C . Since $p=15 > 11$, we swap the two and 15 becomes the right end point of node C . The smaller of left end points of C 's children is 4. Since this is smaller than the current $p=11$, it is moved into node C as this node's left end point. We now

position ourselves at node D . First, we swap $p=11$ and D 's right end point. Now, since D has no children, the current $p=7$ is inserted into node D as D 's left end point. Figure 9.45 shows the result.



Figure 9.45: The interval heap of Figure 9.44 with minimum element removed

The max element may be removed using an analogous procedure.

9.7.4 Initializing an Interval Heap

Interval heaps may be initialized using a strategy similar to that used to initialize ordinary heaps—work your way from the heap bottom to the root ensuring that each subtree is an interval heap. For each subtree, first order the elements in the root; then reinsert the left end point of this subtree's root using the reinsertion strategy used for the *DeleteMin* operation, then reinsert the right end point of this subtree's root using the strategy used for the *DeleteMax* operation.

9.7.5 Complexity of Interval Heap Operations

The operations *GetMin()* and *GetMax()* take $O(1)$ time each; *Insert(x)*, *DeleteMin()*, and *DeleteMax()* take $O(\log n)$ each; and initializing an n element interval heap takes $O(n)$ time.

9.7.6 The Complementary Range Search Problem

In the *complementary range search problem*, we have a *dynamic collection* (i.e., points are added and removed from the collection as time goes on) of one-dimensional points (i.e., points have only an x -coordinate associated with them) and we are to answer queries of the form: what are the points outside of the interval $[a, b]$? For example, if the point collection is 3, 4, 5, 6, 8, 12, the points outside the range $[3, 7]$ are 3, 4, 8, 12.

When an interval heap is used to represent the point collection, a new point can be inserted or an old one removed in $O(\log n)$ time, where n is the number of points in the collection. Note that given the location of an arbitrary element in an interval heap, this element can be removed from the interval heap in $O(\log n)$ time using an algorithm similar to that used to remove an arbitrary element from a heap.

The complementary range query can be answered in $\Theta(k)$ time, where k is the number of points outside the range $[a, b]$. This is done using the following recursive procedure:

- Step 1:** If the interval tree is empty, **return**.
- Step 2:** If the root interval is contained in $[a, b]$, then all points are in the range (therefore, there are no points to report), **return**.
- Step 3:** Report the end points of the root interval that are not in the range $[a, b]$.
- Step 4:** Recursively search the left subtree of the root for additional points that are not in the range $[a, b]$.
- Step 5:** Recursively search the right subtree of the root for additional points that are not in the range $[a, b]$.
- Step 6:** **return**.

Let us try this procedure on the interval heap of Figure 9.44. The query interval is $[4, 32]$. We start at the root. Since the root interval is not contained in the query interval, we reach step 3 of the procedure. Whenever step 3 is reached, we are assured that at least one of the end points of the root interval is outside the query interval. Therefore, each time step 3 is reached, at least one point is reported. In our example, both points 2 and 40 are outside the query interval and are reported. We then search the left and right subtrees of the root for additional points. When the left subtree is searched, we again determine that the root interval is not contained in the query interval. This time only one of the root interval points (i.e., 3) is outside the query range. This point is reported and we proceed to search the left and right subtrees of B for additional points outside the query range. Since the interval of the left child of B is contained in the query range, the

left subtree of B contains no points outside the query range. We do not explore the left subtree of B further. When the right subtree of B is searched, we report the left end point 3 of node C and proceed to search the left and right subtrees of C . Since the intervals of the roots of each of these subtrees is contained in the query interval, these subtrees are not explored further. Finally, we examine the root of the right subtree of the overall tree root, that is the node with interval $[4, 32]$. Since this node's interval is contained in the query interval, the right subtree of the overall tree is not searched further.

We say that a node is *visited* if its interval is examined in Step 2. With this definition of *visited*, we see that the complexity of the above six step procedure is $\Theta(\text{number of nodes visited})$. The nodes visited in the preceding example are the root and its two children, the two children of node B , and the two children of node C . So, 7 nodes are visited and a total of 4 points are reported.

We show that the total number of interval heap nodes visited is at most $3k+1$, where k is the number of points reported. If a visited node reports one or two points, give the node a count of one. If a visited node reports no points, give it a count of zero and add one to the count of its parent (unless the node is the root and so has no parent). The number of nodes with a nonzero count is at most k . Since no node has a count more than 3, the sum of the counts is at most $3k$. Accounting for the possibility that the root reports no point, we see that the number of nodes visited is at most $3k+1$. Therefore, the complexity of the search is $\Theta(k)$. This complexity is asymptotically optimal because every algorithm that reports k points must spend at least $\Theta(1)$ time per reported point.

In our example search, the root gets a count of 2 (1 because it is visited and reports at least one point and another 1 because its right child is visited but reports no point), node B gets a count of 2 (1 because it is visited and reports at least one point and another 1 because its left child is visited but reports no point), and node C gets a count of 3 (1 because it is visited and reports at least one point and another 2 because its left and right children are visited and neither reports a point). The count for each of the remaining nodes in the interval heap is 0.

EXERCISES

1. Start with an empty interval heap and insert the elements 20, 10, 40, 3, 2, 7, 60, 1 and 80 (in this order) using the insertion algorithm developed in this section. Draw the interval heap following each insert.
2. Perform 3 delete-min operations on the interval heap of Figure 9.45. Use the delete min strategy described in this section. Draw the interval heap following each delete min.
3. Perform 4 delete max operations on the interval heap of Figure 9.45. Adapt the delete min strategy of this section to the delete max operation. Draw the interval heap following each delete max operation.

4. Develop the code for all functions of the class *IntervalHeap*, which implements the interval heap data structure and derives from the virtual class *DEPQ*. In addition to the functions specified in *DEPQ* you also must code the initialization function and a function for the complementary range search operation. Test all functions using your own test data.
5. The min-max heap is an alternative heap inspired data structure for the representation of a *DEPQ*. A *min-max heap* is a complete binary tree in which each node has exactly one element. Alternating levels of this tree are min levels and max levels, respectively. The root is on a min level. Let x be any node in a min-max heap. If x is on a min (max) level then the element in x has the minimum (maximum) priority from among all elements in the subtree with root x . A node on a min (max) level is called a *min* (max) node. Figure 9.46 shows an example 13-elements min-max heap. We use shaded circles for max nodes and unshaded circles for min nodes.

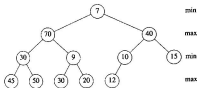


Figure 9.46: A 13-element min-max heap

Fully code and test the class *MinMaxHeap*, which implements a min-max heap using a one-dimensional array for the complete binary tree. Your class must provide an implementation for all *DEPQ* functions. The complexity of *GetMin* and *GetMax* should be $O(1)$ and that for the remaining *DEPQ* functions should be $O(\log n)$.

9.8 REFERENCES AND SELECTED READINGS

Height-biased leftist trees were invented by C. Crane. See, *Linear Lists and Priority Queues as Balanced Binary Trees*, Technical report CS-73-259, Computer Science Dept., Stanford University, Palo Alto, CA, 1972. Weight-biased leftist trees were developed in "Weight biased leftist trees and modified skip lists," S. Cho and S. Sahni, *ACM Jr. on Experimental Algorithms*, Article 2, 1998.

The exercise on lazy deletion is from "Finding minimum spanning trees," by D. Cheriton and R. Tarjan, *SIAM Journal on Computing*, 5, 1976, pp. 724-742.

B-heaps and F-heaps were invented by M. Fredman and R. Tarjan. Their work is reported in the paper "Fibonacci heaps and their uses in improved network optimization algorithms," *JACM*, 34:3, 1987, pp. 596-615. This paper also describes several variants of the basic F-heap as discussed here, as well as the application of F-heaps to the assignment problem and to the problem of finding a minimum-cost spanning tree. Their result is that using F-heaps, minimum-cost spanning trees can be found in $O(e\beta(e,n))$ time, where $\beta(e,n) \leq \log^*n$ when $e \geq n$, $\log^*n = \min\{i \mid \log^{(i)}n \leq 1\}$, $\log^{(0)}n = n$, and $\log^{(i)}n = \log(\log^{(i-1)}n)$. The complexity of finding minimum-cost spanning trees has been further reduced to $O(e \log \beta(e,n))$. The reference for this is "Efficient algorithms for finding minimum spanning trees in undirected and directed graphs," by H. Gabow, Z. Galil, T. Spencer, and R. Tarjan, *Combinatorica*, 6:2, 1986, pp. 109-122.

Pairing heaps were developed in the paper "The pairing heap: A new form of self-adjusting heap", by M. Fredman, R. Sedgwick, R. Sleator, and R. Tarjan, *Algorithmica*, 1, 1986, pp. 111-129. This paper together with "New upper bounds for pairing heaps," by J. Iacono, *Scandinavian Workshop on Algorithm Theory*, LNCS 1831, 2000, pp. 33-42 establishes the amortized complexity of the pairing heap operations. The paper "On the efficiency of pairing heaps and related data structures," by M. Fredman, Jr. of the *ACM*, 46, 1999, pp. 473-501 provides an information theoretic proof that $O(\log \log n)$ is a lower bound on the amortized complexity of the decrease key operation for pairing heaps.

Experimental studies conducted by Skisko and Vitter reported in their paper "Pairing heaps: Experiments and analysis," *Communications of the ACM*, 30, 3, 1987, 234-249 establish the superiority of two pass pairing heaps over multipass pairing heaps. This paper also proposes a variant of pairing heaps (called *auxiliary two pass pairing heaps*) that performs better than two pass pairing heaps. Mower and Shapiro establish the superiority of pairing heaps over Fibonacci heaps, when implementing Prim's minimum spanning tree algorithm, in their paper "An empirical analysis of algorithms for constructing a minimum cost spanning tree," *Second Workshop on Algorithms and Data Structures*, 1991, pp. 400-411.

A large number of data structures, inspired by the fundamental heap structure of Section 5.6, have been developed for the representation of a DEPQ. The symmetric min-max heap was developed in "Symmetric min-max heap: A simpler data structure for double-ended priority queue," by A. Arvind and C. Pandu Rangan, *Information Processing Letters*, 69, 1999, 197-199.

The twin heaps of Williams, the min-max pair heapsof Olariu et al., the interval heaps of Ding and Weiss and van Leeuwen et al., and the diamond deque of Chang and Du are virtually identical data structures. The relevant papers are: "Diamond deque: A simple data structure for priority dequeues," by S. Chang and M. Du, *Information Processing Letters*, 46, 231-237, 1993; "On the Complexity of Building an Interval Heap," by Y. Ding and M. Weiss, *Information Processing Letters*, 50, 143-144, 1994; "Interval heaps," by J. van Leeuwen and D. Wood, *The Computer Journal*, 36, 3, 209-216, 1993; "A mergeable double-ended priority queue," by S. Olariu, C. Overstreet, and Z. Wen, *The Computer Journal*, 34, 5, 423-427, 1991; and "Algorithm 233," by J. Williams, *Communications of the ACM*, 7, 347-348, 1964.

The min-max heap and deap are additional heap-inspired structures for DEPQs. These data structures were developed in "Min-max heaps and generalized priority queues," by M. Atkinson, J. Sack, N. Santoro, and T. Strothotte, *Communications of the ACM*, 29:10, 1986, pp. 996-1000 and "The deap: A double-ended heap to implement double-ended priority queues," by S. Carlsson, *Information Processing Letters*, 26, 1987, pp. 33-36, respectively.

Data structures for meldable DEPQs are developed in "The relaxed min-max heap: A mergeable double-ended priority queue," by Y. Ding and M. Weiss, *Acta Informatica*, 30, 215-231, 1993; "Fast meldable priority queues," by G. Brodal, *Workshop on Algorithms and Data Structures*, 1995 and "Mergeable double ended priority queue," by S. Cho and S. Sahni, *International Journal on Foundation of Computer Sciences*, 10, 1, 1999, 1-18.

General techniques to arrive at a data structure for a DEPQ from one for a single-ended priority queue are developed in "Correspondence based data structures for double ended priority queues," by K. Chong and S. Sahni, *ACM Jr. on Experimental Algorithmics*, Volume 5, 2000, Article 2.

For more on priority queues, see Chapters 5 through 8 of "Handbook of data structures and applications," edited by D. Mehta and S. Sahni, Chapman & Hall/CRC, Boca Raton, 2005.

CHAPTER 10

Efficient Binary Search Trees

10.1 OPTIMAL BINARY SEARCH TREES

Binary search trees were introduced in Chapter 5. In this section, we consider the construction of binary search trees for a static set of elements. That is, we make no additions to or deletions from the set. Only searches are performed.

A sorted list can be searched using a binary search. For this search, we can construct a binary search tree with the property that searching this tree using the function *Get* (Program 5.19) is equivalent to performing a binary search on the sorted list. For instance, a binary search on the sorted element list (5, 10, 15) (for convenience, all examples in this chapter show only an element's key rather than the complete element) corresponds to using algorithm *Get* on the binary search tree of Figure 10.1. Although this tree is a full binary tree, it may not be the optimal binary search tree to use when the probabilities with which different elements are searched are different.

To find an optimal binary search tree for a given collection of elements, we



Figure 10.1: Binary search tree corresponding to a binary search on the list (5, 10, 15)

must first decide on a cost measure for search trees. When searching for an element at level i , function *Get* makes i iterations of the *for* loop. Since this *for* loop determines the cost of the search, it is reasonable to use the level number of a node as its cost.

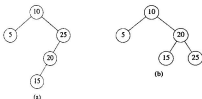


Figure 10.2: Two binary search trees

Example 10.1: Consider the two search trees of Figure 10.2. The second of these requires at most three comparisons to decide whether the element being sought is in the tree. The first binary tree may require four comparisons, since any search key k such that $10 < k < 20$ will test four nodes. Thus, as far as worst-case search time is concerned, the second binary tree is more desirable

than the first. To search for a key in the first tree takes one comparison for the 10, two for each of 5 and 25, three for 20, and four for 15. Assuming that each key is searched for with equal probability, the average number of comparisons for a successful search is 2.4. For the second binary search tree this amount is 2.2. Thus, the second tree has a better average behavior, too.

Suppose that each of 5, 10, 15, 20 and 25 is searched for with probability 0.3, 0.3, 0.05, 0.05 and 0.3, respectively. The average number of comparisons for a successful search in the trees of Figure 10.2 (a) and (b) is 1.85 and 2.05, respectively. Now, the first tree has better average behavior than the second tree! \square

In evaluating binary search trees, it is useful to add a special "square" node at every null link. Doing this to the trees of Figure 10.2 yields the trees of Figure 10.3. Remember that every binary tree with n nodes has $n + 1$ null links and therefore will have $n + 1$ square nodes. We shall call these nodes *external* nodes because they are not part of the original tree. The remaining nodes will be called *internal* nodes. Each time a binary search tree is examined for an identifier that is not in the tree, the search terminates at an external node. Since all such searches are unsuccessful searches, external nodes will also be referred to as *failure nodes*. A binary tree with external nodes added is an *extended binary tree*. The concept of an extended binary tree as just defined is the same as that defined in connection with leftist trees in Chapter 9. Figure 10.3 shows the extended binary trees corresponding to the search trees of Figure 10.2.

We define the *external path length* of a binary tree to be the sum over all external nodes of the lengths of the paths from the root to those nodes. Analogously, the *internal path length* is the sum over all internal nodes of the lengths of the paths from the root to those nodes. The internal path length, I , for the tree of Figure 10.3(a) is

$$I = 0 + 1 + 1 + 2 + 3 = 7$$

Its external path length, E , is

$$E = 2 + 2 + 4 + 4 + 3 + 2 = 17$$

Exercise 1 of this section shows that the internal and external path lengths of a binary tree with n internal nodes are related by the formula $E = I + 2n$. Hence, binary trees with the maximum E also have maximum I . Over all binary trees with n internal nodes, what are the maximum and minimum possible values for I ? The worst case, clearly, is when the tree is skewed (i.e., when the tree has a depth of n). In this case,

$$I = \sum_{i=0}^{n-1} i = n(n-1)/2$$

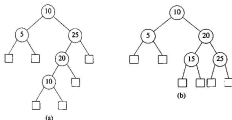


Figure 10.3: Extended binary trees corresponding to search trees of Figure 10.2

To obtain trees with minimal I , we must have as many internal nodes as close to the root as possible. We can have at most 2 nodes at distance 1, 4 at distance 2, and in general, the smallest values for I is

$$0 + 2 \cdot 1 + 4 \cdot 2 + 8 \cdot 3 + \cdots +$$

One tree with minimal internal path length is the complete binary tree defined in Section 5.2. If we number the nodes in a complete binary tree as in Section 5.2, then we see that the distance of node i from the root is $\lfloor \log_2 i \rfloor$. Hence, the smallest value for I is

$$\sum_{1 \leq i \leq n} \lfloor \log_2 i \rfloor = O(n \log_2 n)$$

Let us now return to our original problem of representing a static element set as a binary search tree. Let a_1, a_2, \dots, a_n with $a_1 < a_2 < \dots < a_n$ be the element keys. Suppose that the probability of searching for each a_i is p_i . The total cost of any binary search tree for this set of keys is

$$\sum_{1 \leq i \leq n} p_i \cdot \text{level}(a_i)$$

when only successful searches are made. Since unsuccessful searches (i.e., searches for keys not in the table) will also be made, we should include the cost of these searches in our cost measure, too. Unsuccessful searches terminate with algorithm *Get* returning a 0 pointer (Program 5.19). Every node with an empty subtree defines a point at which such a termination can take place. Let us replace every empty subtree by a failure node. The keys not in the binary search tree may be partitioned into $n + 1$ classes E_i , $0 \leq i \leq n$. E_0 contains all keys X such that $X < a_1$. E_i contains all keys X such that $a_i < X < a_{i+1}$, $1 \leq i < n$, and E_n contains all keys X , $X > a_n$. It is easy to see that for all keys in a particular class E_i , the search terminates at the same failure node, and it terminates at different failure nodes for keys in different classes. The failure nodes may be numbered 0 to n , with i being the failure node for class E_i , $0 \leq i \leq n$. If q_i is the probability that the key being sought is in E_i , then the cost of the failure nodes is

$$\sum_{0 \leq i \leq n} q_i \cdot (\text{level}(\text{failure node } i) - 1)$$

Therefore, the total cost of a binary search tree is

$$\sum_{1 \leq i \leq n} p_i \cdot \text{level}(a_i) + \sum_{0 \leq i \leq n} q_i \cdot (\text{level}(\text{failure node } i) - 1) \quad (10.1)$$

An optimal binary search tree for a_1, \dots, a_n is one that minimizes Eq. (10.1) over all possible binary search trees for this set of keys. Note that since all searches must terminate either successfully or unsuccessfully, we have

$$\sum_{1 \leq i \leq n} p_i + \sum_{0 \leq i \leq n} q_i = 1$$

Example 10.2: Figure 10.4 shows the possible binary search trees for the key set $(a_1, a_2, a_3) = (5, 10, 15)$. With equal probabilities, $p_i = q_j = 1/7$ for all i and j , we have

$$\begin{aligned} \text{cost}(\text{tree } a) &= 15/7; & \text{cost}(\text{tree } b) &= 13/7 \\ \text{cost}(\text{tree } c) &= 15/7; & \text{cost}(\text{tree } d) &= 15/7 \\ \text{cost}(\text{tree } e) &= 15/7 \end{aligned}$$

As expected, tree *b* is optimal. With $p_1 = 0.5$, $p_2 = 0.1$, $p_3 = 0.05$, $q_0 = 0.15$, $q_1 = 0.1$, $q_2 = 0.05$, and $q_3 = 0.05$ we have

$$\begin{aligned} \text{cost}(\text{tree } a) &= 3.65; & \text{cost}(\text{tree } b) &= 1.9 \\ \text{cost}(\text{tree } c) &= 1.5; & \text{cost}(\text{tree } d) &= 2.05 \\ \text{cost}(\text{tree } e) &= 1.6 \end{aligned}$$

Tree *c* is optimal with this assignment of p 's and q 's. \square

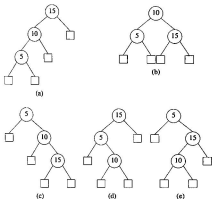


Figure 10.4: Binary search trees with three elements.

How does one determine the optimal binary search tree? We could proceed as in Example 10.2 and explicitly generate all possible binary search trees, then compute the cost of each tree, and determine the tree with minimum cost. Since the cost of an n -node binary search tree can be determined in $O(n)$ time, the complexity of the optimal binary search tree algorithm is $O(nN(n))$, where $N(n)$ is the number of distinct binary search trees with n keys. From Section 5.11 we know that $N(n) = O(4^n/n^{3/2})$. Hence, this brute-force algorithm is impractical for large n . We can find a fairly efficient algorithm by making some observations about the properties of optimal binary search trees.

Let $a_1 < a_2 < \cdots < a_n$ be the n keys to be represented in a binary search tree. Let T_{ij} denote an optimal binary search tree for a_{i+1}, \dots, a_j , $i < j$. By

convention T_{ij} is an empty tree for $0 \leq i \leq n$, and T_{ij} is not defined for $i > j$. Let c_{ij} be the cost of the search tree T_{ij} . By definition c_{ij} will be 0. Let r_{ij} be the root of T_{ij} , and let

$$w_{ij} = q_i + \sum_{k=i+1}^j (q_k + p_k)$$

be the weight of T_{ij} . By definition $r_{ij} = 0$, and $w_{ij} = q_i$, $0 \leq i \leq n$. Therefore, T_{0n} is an optimal binary search tree for a_1, \dots, a_n . Its cost is c_{0n} , its weight is w_{0n} , and its root is r_{0n} .

If T_{ij} is an optimal binary search tree for a_{i+1}, \dots, a_j , and $r_{ij} = k$, then k satisfies the inequality $i < k \leq j$. T_{ij} has two subtrees L and R . L is the left subtree and contains the keys a_{i+1}, \dots, a_{k-1} , and R is the right subtree and contains the keys a_{k+1}, \dots, a_j (Figure 10.5). The cost c_{ij} of T_{ij} is

$$c_{ij} = p_k + \text{cost}(L) + \text{cost}(R) + \text{weight}(L) + \text{weight}(R) \quad (10.2)$$

where $\text{weight}(L) = \text{weight}(T_{i,k-1}) = w_{i,k-1}$, and $\text{weight}(R) = \text{weight}(T_{k,j}) = w_{k,j}$.



Figure 10.5: An optimal binary search tree T_{ij}

From Eq. (10.2) it is clear that if c_{ij} is to be minimal, then $\text{cost}(L) = c_{i,k-1}$ and $\text{cost}(R) = c_{k,j}$, or otherwise we could replace either L or R by a subtree with a lower cost, thus getting a binary search tree for a_{i+1}, \dots, a_j with a lower cost than c_{ij} . This violates the assumption that T_{ij} is optimal. Hence, Eq. (10.2) becomes

$$\begin{aligned} c_{ij} &= p_k + c_{i,k-1} + c_{k,j} + w_{i,k-1} + w_{k,j} \\ &= w_{ij} + c_{i,k-1} + c_{k,j} \end{aligned} \quad (10.3)$$

Since T_{ij} is optimal, it follows from Eq. (10.3) that $r_{ij} = k$ is such that

$$w_{ij} + c_{i,k-1} + c_{k,j} = \min_{i < k \leq j} \{w_{ij} + c_{i,k-1} + c_{k,j}\}$$

or

$$c_{i,j-1} + c_{ij} = \min_{i \leq k < j} [c_{i,k-1} + c_{kj}] \quad (10.4)$$

Equation (10.4) gives us a means of obtaining T_{0n} and c_{0n} , starting from the knowledge that $T_{i0} = \phi$ and $c_{i0} = 0$.

Example 10.3: Let $n = 4$ and $(a_1, a_2, a_3, a_4) = (10, 15, 20, 25)$. Let $(p_1, p_2, p_3, p_4) = (3, 3, 1, 1)$ and $(q_1, q_2, q_3, q_4) = (2, 3, 1, 1)$. The p 's and q 's have been multiplied by 16 for convenience. Initially, $w_0 = q_1$, $c_{i0} = 0$, and $r_{i0} = 0$, $0 \leq i \leq 4$. Using Eqs. (10.3) and (10.4), we get

$$\begin{aligned} w_{01} &= p_1 + w_{00} + w_{11} = p_1 + q_1 + w_{00} = 8 \\ c_{01} &= w_{01} + \min\{c_{00} + c_{11}\} = 8 \\ r_{01} &= 1 \\ w_{12} &= p_2 + w_{11} + w_{22} = p_2 + q_2 + w_{11} = 7 \\ c_{12} &= w_{12} + \min\{c_{11} + c_{22}\} = 7 \\ r_{12} &= 2 \\ w_{23} &= p_3 + w_{22} + w_{33} = p_3 + q_3 + w_{22} = 3 \\ c_{23} &= w_{23} + \min\{c_{22} + c_{33}\} = 3 \\ r_{23} &= 3 \\ w_{34} &= p_4 + w_{33} + w_{44} = p_4 + q_4 + w_{33} = 3 \\ c_{34} &= w_{34} + \min\{c_{33} + c_{44}\} = 3 \\ r_{34} &= 4 \end{aligned}$$

Knowing $w_{i,j+1}$ and $c_{i,j+1}$, $0 \leq i < 4$, we can use Eqs. (10.3) and (10.4) again to compute $w_{i,j+2}$, $c_{i,j+2}$, $r_{i,j+2}$, $0 \leq i < 3$. This process may be repeated until w_{04} , c_{04} , and r_{04} are obtained. The table of Figure 10.6 shows the results of this computation. From the table, we see that $c_{04} = 32$ is the minimal cost of a binary search tree for a_1 to a_4 . The root of tree T_{04} is a_2 . Hence, the left subtree is T_{01} and the right subtree T_{34} . T_{01} has root a_1 and subtrees T_{00} and T_{11} . T_{34} has root a_3 ; its left subtree is therefore T_{23} and right subtree T_{34} . Thus, with the data in the table it is possible to reconstruct T_{04} . Figure 10.7 shows T_{04} . \square

Example 10.3 illustrates how Eq. (10.4) may be used to determine the c 's and r 's, as well as how to reconstruct T_{0n} knowing the r 's. Let us examine the complexity of this function to evaluate the c 's and r 's. The evaluation function described in Example 10.3 requires us to compute c_{ij} for $(j-i) = 1, 2, \dots, n$ in

	0	1	2	3	4
0	$w_{00} = 2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 1$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$	
2	$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$		
3	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 2$	$w_{14} = 11$ $c_{14} = 19$ $r_{14} = 2$			
4	$w_{04} = 16$ $c_{04} = 32$ $r_{04} = 2$				

Figure 10.6: Computation of c_{ij} and r_{ij} . The computation is carried out by row from row 0 to row 4



Figure 10.7: Optimal binary search tree for Example 10.3

562 Efficient Binary Search Trees

that order. When $j - i = m$, there are $n - m + 1$ c_{ij} 's to compute. The computation of each of these c_{ij} 's requires us to find the minimum of m quantities (see Eq. (10.4)). Hence, each such c_{ij} can be computed in time $O(m)$. The total time for all c_{ij} 's with $j - i = m$ is therefore $O(m(n - m))$. The total time to evaluate all the c_{ij} 's and r_{ij} 's is

$$\sum_{1 \leq i < j \leq n} (nm - m^2) = O(n^3)$$

Actually we can do better than this using a result due to D. E. Knuth that states that the optimal i in Eq. (10.4) may be found by limiting the search to the range $r_{i,j-1} \leq i \leq r_{i+1,j}$. In this case, the computing time becomes $O(n^2)$ (see Exercise 3). Function *Obst* (Program 10.1) uses this result to obtain in $O(n^2)$ time the values of w_{ij} , r_{ij} , and c_{ij} , $0 \leq i \leq j \leq n$. The actual tree $T_{0,n}$ may be constructed from the values of r_{ij} in $O(n)$ time. The algorithm for this is left as an exercise.

Function *Obst* (Program 10.2) computes the cost $c[i][j] = c_{ij}$ of optimal binary search trees T_{ij} for keys a_{i+1}, \dots, a_j . It also computes $r[i][j] = r_{ij}$, the root of T_{ij} . $w[i][j] = w_{ij}$ is the weight of T_{ij} . The two-dimensional arrays c , r and w are global arrays of type `int`. The inputs to this function are the success and failure probability arrays, $p[]$ and $q[]$ and the number of keys n . The array elements $p[0]$ and $q[0]$ are not used.

EXERCISES

- (a) Prove by induction that if T is a binary tree with s internal nodes, I its internal path length, and E its external path length, then $E = I + 2s$, $s \geq 0$.
- (b) Using the result of (a), show that the average number of comparisons s in a successful search is related to the average number of comparisons, u , in an unsuccessful search by the formula

$$s = (1 + 1/s)u - 1, \quad s \geq 1$$

- Use function *Obst* (Program 10.1), to compute w_{ij} , r_{ij} , and c_{ij} , $0 \leq i < j \leq 4$, for the key set $(a_1, a_2, a_3, a_4) = (3, 10, 15, 30)$, with $p_1 = 1/20$, $p_2 = 1/5$, $p_3 = 1/10$, $p_4 = 1/20$, $q_0 = 1/5$, $q_1 = 1/10$, $q_2 = 1/5$, $q_3 = 1/20$, and $q_4 = 1/20$. Using the r_{ij} 's, construct the optimal binary search tree.
- (a) Complete function *Obst* by providing the code for function *KnuthMin*.
- (b) Show that the computing time complexity of *Obst* is $O(n^2)$.
- (c) Write a C++ function *BSTConstruct* to construct the optimal binary

```

void Obstr(double *p, double *q, int n)
{
    for (int i = 0; i < n; i++) {
        w[i][i] = q[i]; r[i][i] = c[i][i] = 0;    // initialize
        w[i][i+1] = q[i] + q[i+1] + p[i+1];    // optimal trees with one node
        r[i][i+1] = i + 1;
        c[i][i+1] = w[i][i+1];
    }
    w[n][n] = q[n]; r[n][n] = c[n][n] = 0;
    for (int m = 2; m <= n; m++) // find optimal trees with m nodes
        for (i = 0; i <= n - m; i++)
        {
            int j = i + m;
            w[i][j] = w[i][j-1] + p[j] + q[j];
            int k = KnuthMin(i, j);
            // KnuthMin returns a value k in the range [r[i][j-1], r[i+1][j]]
            // minimizing c[i][k-1] + c[k][j]
            c[i][j] = w[i][j] + c[i][k-1] + c[k][j]; // Eq. (10.3)
            r[i][j] = k;
        }
    } // end of Obstr
}

```

Program 10.1: Finding an optimal binary search tree

- search tree T_{Opt} given the costs r_{ij} , $0 \leq i < j \leq n$. Show that this can be done in time $O(n)$.
4. Implement in C++ a BST constructor that constructs an optimal binary search tree, given the success and failure probability arrays $p[]$ and $q[]$, the array of keys $a[]$, and the number of keys n .
 5. Since, often, only the approximate values of the p 's and q 's are known, it is perhaps just as meaningful to find a binary search tree that is nearly optimal (i.e., its cost, Eq. (10.1), is almost minimal for the given p 's and q 's). This exercise explores an $O(n \log n)$ algorithm that results in nearly optimal binary search trees. The search tree heuristic we shall study is
Choose the root a_k such that $w_{a_{k+1}, i} = w_{a_{k-1}, j}$ is as small as possible. Repeat this process to find the left and right subtrees of a_k .
 - (a) Using this heuristic obtain the resulting binary search tree for the data of Exercise 2. What is its cost?

- (b) Write a C++ function implementing the above heuristic. The time complexity of your function should be $O(n \log n)$.

An analysis of the performance of this heuristic may be found in the paper by Mehlhorn (see the References and Selected Readings section).

10.2 AVL TREES

Dynamic collections of elements may also be maintained as binary search trees. In Chapter 5, we saw how insertions and deletions can be performed on binary search trees. Figure 10.8 shows the binary search tree obtained by entering the months JANUARY to DECEMBER in that order into an initially empty binary search tree by using function *Insert* (Program 5.21).



Figure 10.8: Binary search tree obtained for the months of the year

The maximum number of comparisons needed to search for any key in the tree of Figure 10.8 is six for NOVEMBER. The average number of comparisons is $(1 \text{ for JANUARY} + 2 \text{ each for FEBRUARY and MARCH} + 3 \text{ each for APRIL, JUNE and MAY} + \dots + 6 \text{ for NOVEMBER})/12 = 42/12 = 3.5$. If the

months are entered in the order JULY, FEBRUARY, MAY, AUGUST, DECEMBER, MARCH, OCTOBER, APRIL, JANUARY, JUNE, SEPTEMBER, NOVEMBER, then the tree of Figure 10.9 is obtained.



Figure 10.9: A balanced tree for the months of the year

The tree of Figure 10.9 is well balanced and does not have any paths to a leaf node that are much longer than others. This is not true of the tree of Figure 10.8, which has six nodes on the path from the root to NOVEMBER and only two nodes on the path to APRIL. Moreover, during the construction of the tree of Figure 10.9, all intermediate trees obtained are also well balanced. The maximum number of key comparisons needed to find any key is now 4, and the average is $37/12 = 3.1$. If the months are entered in lexicographic order, instead, the tree degenerates to a chain as in Figure 10.10. The maximum search time is now 12 key comparisons, and the average is 6.5. Thus, in the worst case, searching a binary search tree corresponds to sequential searching in a sorted linear list. When the keys are entered in a random order, the tree tends to be balanced as in Figure 10.9. If all permutations are equally probable, then the average search and insertion time is $O(\log n)$ for an n -node binary search tree.

From our earlier study of binary trees, we know that both the average and maximum search time will be minimized if the binary search tree is maintained as a complete binary tree at all times. However, since we are dealing with a dynamic situation, it is difficult to achieve this ideal without making the time required to insert a key very high. This is so because in some cases it would be necessary to restructure the whole tree to accommodate the new entry and at the same time have a complete binary search tree. It is, however, possible to keep the tree balanced to ensure both an average and worst-case search time of

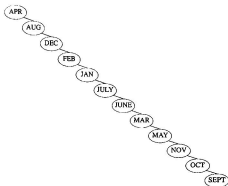


Figure 10.10: Degenerate binary search tree

$O(\log n)$ for a tree with n nodes. In this section, we study one method of growing balanced binary trees. These balanced trees will have satisfactory search, insertion and deletion time properties. Other ways to maintain balanced search trees are studied in later sections.

In 1962, Adelson-Velskii and Landis introduced a binary tree structure that is balanced with respect to the heights of subtrees. As a result of the balanced nature of this type of tree, dynamic retrievals can be performed in $O(\log n)$ time if the tree has n nodes in it. At the same time, a new key can be entered or deleted from such a tree in time $O(\log n)$. The resulting tree remains height-balanced. This tree structure is called an AVL tree. As with binary trees, it is natural to define AVL trees recursively.

Definition: An empty tree is height-balanced. If T is a nonempty binary tree with T_L and T_R as its left and right subtrees respectively, then T is *height-balanced* iff (1) T_L and T_R are height-balanced and (2) $|h_L - h_R| \leq 1$ where h_L and h_R are the heights of T_L and T_R , respectively. \square

The definition of a height-balanced binary tree requires that every subtree also be height-balanced. The binary tree of Figure 10.8 is not height-balanced, since the height of the left subtree of the tree with root APRIL is 0 and that of the right subtree is 2. The tree of Figure 10.9 is height-balanced while that of Figure 10.10 is not. To illustrate the processes involved in maintaining a height-balanced binary search tree, let us try to construct such a tree for the months of the year. This time let us assume that the insertions are made in the following order: MARCH, MAY, NOVEMBER, AUGUST, APRIL, JANUARY, DECEMBER, JULY, FEBRUARY, JUNE, OCTOBER, SEPTEMBER. Figure 10.11 shows the tree as it grows and the restructuring involved in keeping the tree balanced. The numbers above each node represent the difference in heights between the left and right subtrees of that node. This number is referred to as the *balance factor* of the node.

Definition: The *balance factor*, $BF(T)$, of a node T in a binary tree is defined to be $h_L - h_R$, where h_L and h_R , respectively, are the heights of the left and right subtrees of T . For any node T in an AVL tree, $BF(T) = -1, 0$, or 1 . \square

Inserting MARCH and MAY results in the binary search trees (a) and (b) of Figure 10.11. When NOVEMBER is inserted into the tree, the height of the right subtree of MARCH becomes 2, whereas that of the left subtree is 0. The tree has become unbalanced. To rebalance the tree, a rotation is performed. MARCH is made the left child of MAY, and MAY becomes the root (Figure 10.11(c)). The introduction of AUGUST leaves the tree balanced (Figure 10.11(d)).

The next insertion, APRIL, causes the tree to become unbalanced again. To rebalance the tree, another rotation is performed. This time, it is a clockwise rotation. MARCH is made the right child of AUGUST, and AUGUST becomes the root of the subtree (Figure 10.11(e)). Note that both of the previous rotations were carried out with respect to the closest parent of the new node that had a balance factor of ± 2 . The insertion of JANUARY results in an unbalanced tree. This time, however, the rotation involved is somewhat more complex than in the earlier situations. The common point, however, is that the rotation is still carried out with respect to the nearest parent of JANUARY with a balance factor ± 2 . MARCH becomes the new root. AUGUST, together with its left subtree, becomes the left subtree of MARCH. The left subtree of MARCH becomes the right subtree of AUGUST. MAY and its right subtree, which three keys greater than MARCH, become the right subtree of MARCH. (If MARCH had had a

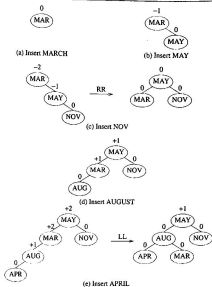


Figure 18.11: Balanced trees obtained for the months of the year (continued on next page)

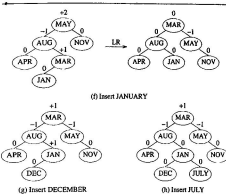


Figure 10.11: Balanced trees obtained for the months of the year (continued on next page)

nonempty right subtree, this could have become the left subtree of MAY, since all keys would have been less than MAY.)

Inserting DECEMBER and JULY necessitates no rebalancing. When FEBRUARY is inserted, the tree becomes unbalanced again. The rebalancing process is very similar to that used when JANUARY was inserted. The nearest parent with balance factor ± 2 is AUGUST. DECEMBER becomes the new root of that subtree. AUGUST, with its left subtree, becomes the left subtree. JANUARY, with its right subtree, becomes the right subtree of DECEMBER; FEBRUARY becomes the left subtree of JANUARY. (If DECEMBER had had a left subtree, it would have become the right subtree of AUGUST.) The insertion of JUNE requires the same rebalancing as in Figure 10.11(f). The rebalancing



(i) Insert FEBRUARY



(j) Insert JUNE

Figure 10.11: Balanced trees obtained for the months of the year (continued on next page)

following the insertion of OCTOBER is identical to that following the insertion of NOVEMBER. Inserting SEPTEMBER leaves the tree balanced.

In the preceding example we saw that the addition of a node to a balanced binary search tree could unbalance it. The rebalancing was carried out using four different kinds of rotations: LL, RR, LR, and RL (Figure 10.11 (a), (c), (f), and (j), respectively). LL and RR are symmetric, as are LR and RL. Their

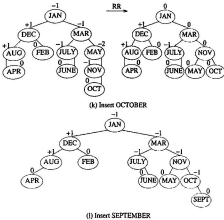


Figure 10.11: Balanced trees obtained for the months of the year

rotations are characterized by the nearest ancestor, A , of the inserted node, Y , whose balance factor becomes ± 2 . The following characterization of rotation types is obtained:

- LL: new node Y is inserted in the left subtree of the left subtree of A
- LR: Y is inserted in the right subtree of the left subtree of A
- RR: Y is inserted in the right subtree of the right subtree of A

572 Efficient Binary Search Trees

RL: Y is inserted in the left subtree of the right subtree of A

A moment's reflection will show that if a height-balanced binary tree becomes unbalanced as a result of an insertion, then these are the only four cases possible for rebalancing. Figures 10.12 and 10.13 show the LL and LR rotations in terms of abstract binary trees. The RR and RL rotations are symmetric. The root node in each of the trees of the figures represents the nearest ancestor whose balance factor has become ± 2 as a result of the insertion. In the example of Figure 10.11 and in the rotations of Figures 10.12 and 10.13, notice that the height of the subtree involved in the rotation is the same after rebalancing as it was before the insertion. This means that once the rebalancing has been carried out on the subtree in question, examining the remaining tree is unnecessary. The only nodes whose balance factors can change are those in the subtree that is rotated.

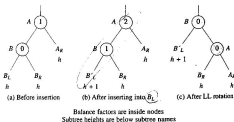


Figure 10.12: An LL rotation

The transformations done to remedy LL and RR imbalances are often called *single rotations*, while those done for LR and RL imbalances are called *double rotations*. The transformation for an LR imbalance can be viewed as an RR rotation followed by an LL rotation, while that for an RL imbalance can be viewed as an LL rotation followed by an RR rotation.

To carry out the rotations of Figures 10.12 and 10.13, it is necessary to locate the node A around which the rotation is to be performed. As remarked

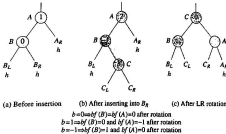


Figure 10.13: An LR rotation

earlier, this is the nearest ancestor of the newly inserted node whose balance factor becomes ± 2 . For a node's balance factor to become ± 2 , its balance factor must have been ± 1 before the insertion. Therefore, before the insertion, the balance factors of all nodes on the path from A to the new insertion point must have been 0. With this information, the node A is readily determined to be the nearest ancestor of the new node having a balance factor ± 1 before insertion. To complete the rotations, the address of F , the parent of A , is also needed. The changes in the balance factors of the relevant nodes are shown in Figures 10.12 and 10.13. Knowing F and A , these changes can be carried out easily.

What happens when the insertion of a node does not result in an unbalanced tree (see Figure 10.11 (a), (b), (d), (g), (h), and (i))? Although no restructuring of the tree is needed, the balance factors of several nodes change. Let A be the nearest ancestor of the new node with balance factor ± 1 before insertion. If, as a result of the insertion, the tree did not get unbalanced, even though some path length increased by 1, it must be that the new balance factor of A is 0. If there is no ancestor A with a balance factor ± 1 (as in Figure 10.11 (a), (b), (d), (g), and (i)), let A be the root. The balance factors of nodes from A to the parent of the new node will change to ± 1 (see Figure 10.11 (h); $A = \text{JANUARY}$). Note that in both cases, the procedure to determine A is the same as when rebalancing

is needed. The remaining details of the insertion-rebalancing process are spelled out in function *Avl::Insert* (Program 10.3). The class definitions in use are

```
template <class K, class E> class AVL;

template <class K, class E>
class AvNode {
friend class AVL<K, E>;
public:
    AvNode(const K& k, const E& e)
        {key = k; element = e; bf = 0; leftChild = rightChild = 0;}
private:
    K key;
    E element;
    int bf;
    AvNode<K, E> *leftChild, *rightChild;
};

template <class K, class E>
class AVL {
public:
    AVL() : root(0) {}
    E& Search(const K& k) const;
    void Insert(const K& k, const E& e);
    void Delete(const K& k);

private:
    AvNode<K, E> *root;
};
```

Here, *E* is the data type of the elements in the AVL tree and *K* is the data type of the keys.

```
template <class K, class E>
void AVL<K, E>::Insert(const K& k, const E& e)
{
    if (!root) { // special case: empty tree
        root = new AvNode<K, E>(k, e);
        return;
    }
    // Phase 1: Locate insertion point for e.
    AvNode<K, E> *a = 0, // most recent node with bf = ±1
        *pa = 0,          // parent of a
        *p = root,        // p moves through the tree
        *pp = 0;          // parent of p
```

```

while(p) {
  if (p→bf) { a = p; pa = pp;
    if (k < p→key) {pp = p; p = p→leftChild; } // take left branch
    else if (k > p→key) {pp = p; p = p→rightChild; }
    else { p→element = x; return; } // k already in tree
  } // end of while

```

//Phase 2: insert and rebalance. *k* is not in the tree and
H may be inserted as the appropriate child of *pp*.

AVLNode <*K*, *E*> **y* = new *AVLNode* <*T*>(*k*, *e*);

if (*k* < *pp*→*key*) *pp*→*leftChild* = *y*; // insert as left child
 else *pp*→*rightChild* = *y*; // insert as right child

//Adjust balance factors of nodes on path from *a* to *pp*. By the definition
B of *a*, all nodes on this path presently have a balance factor of 0. Their new
B balance factor will be ±1. *d* = +1 implies that *k* is inserted in the left subtree
B of *a*. *d* = -1 implies that *k* is inserted in the right subtree of *a*.

int *d*;

AVLNode <*K*, *E*> **b*, // child of *a*
 **c*; // child of *b*

if (*k* > *a*→*key*) { *b* = *p* = *a*→*rightChild*; *d* = -1; }

else { *b* = *p* = *a*→*leftChild*; *d* = 1; }

while (*p* != *y*)

 if (*k* > *p*→*key*) { // *B* height of right increases by 1
 p→*bf* = -1; *p* = *p*→*rightChild*;

 }

 else { // *B* height of left increases by 1

p→*bf* = 1; *p* = *p*→*leftChild*;

 }

// Is tree unbalanced?

if (!(*a*→*bf*) || !(*a*→*bf* + *d*)) { // tree still balanced
 a→*bf* += *d*; return;

}

// tree unbalanced, determine rotation type

if (*d* == 1) { // left imbalance

 if (*b*→*bf* == 1) { // rotation type LL

a→*leftChild* = *b*→*rightChild*;

b→*rightChild* = *a*; *a*→*bf* = 0; *b*→*bf* = 0;

 }

 else { // rotation type LR

c = *b*→*rightChild*;


```

    b->rightChild = c->leftChild;
    a->leftChild = c->rightChild;
    c->leftChild = b;
    c->rightChild = a;
    switch (c->bf) {
        case J:
            a->bf = -1; b->bf = 0;
            break;
        case -J:
            b->bf = 1; a->bf = 0;
            break;
        case 0:
            b->bf = 0; a->bf = 0;
            break;
    }
    c->bf = 0; b = c; // b is the new root
} // end of LR
} // end of left imbalance
else { // right imbalance: this is symmetric to left imbalance
}

// Subtree with root b has been rebalanced.
if (!pa) root = b;
else if (a == pa->leftChild) pa->leftChild = b;
    else pa->rightChild = b;
return;
} // end of AVL::insert

```

Program 10.3: Insertion into an AVL tree

To really understand the insertion algorithm, you should try it out on the example of Figure 10.11. An analysis of the algorithm reveals that if h is the height of the tree before insertion, then the time to insert a new key is $O(h)$. This is the same as for unbalanced binary search trees, although the overhead is significantly greater now. In the case of binary search trees, however, if there were n nodes in the tree, then h could, in the worst case, be n (Figure 10.10), and the worst case insertion time would be $O(n)$. In the case of AVL trees, however, h can be at most $O(\log n)$, so the worst-case insertion time is $O(\log n)$. To see this, let N_h be the minimum number of nodes in a height-balanced tree of height h . In the worst case, the height of one of the subtrees will be $h-1$ and that of the other $h-2$. Both of these subtrees are also height balanced. Hence, $N_h = N_{h-1} + N_{h-2} + 1$, and $N_0 = 0$, $N_1 = 1$ and $N_2 = 2$. Note the similarity

between this recursive definition for N_h and that for the Fibonacci number $F_n = F_{n-1} + F_{n-2}$, $F_0 = 0$, and $F_1 = 1$. In fact, we can show that $N_h = F_{h+2} - 1$ for $h \geq 0$ (see Exercise 3). From Fibonacci number theory it is known that $F_n = \phi^n / \sqrt{5}$, where $\phi = (1 + \sqrt{5})/2$. Hence, $N_h = \phi^{h+2} / \sqrt{5} - 1$. This means that if there are n nodes in the tree, then its height, h , is at most $\log_\phi (\sqrt{5}(n+1)) - 2 = 1.44 \log(n+1) - 0.33$. The worst-case insertion time for a height-balanced tree with n nodes is, therefore, $O(\log n)$.

The exercises show that it is possible to find and delete a node with key X and to find and delete the k th node from a height-balanced tree in $O(\log n)$ time. Results of an empirical study of deletion in height-balanced trees may be found in the paper by Karlton et al. (see the References and Selected Readings section). Their study indicates that a random insertion requires no rebalancing, a rebalancing rotation of type LL or RR, and a rebalancing rotation of type LR and RL, with probabilities 0.5349, 0.2327, and 0.2324, respectively. Figure 10.14 compares the worst-case times of certain operations on sorted sequential lists, sorted linked lists, and AVL trees.

Operation	Sequential list	Linked list	AVL tree
Search for element with key k	$O(\log n)$	$O(n)$	$O(\log n)$
Search for j th item	$O(1)$	$O(j)$	$O(\log n)$
Delete element with key k	$O(n)$	$O(1)^1$	$O(\log n)$
Delete j th element	$O(n - j)$	$O(j)$	$O(\log n)$
Insert	$O(n)$	$O(1)^2$	$O(\log n)$
Output in order	$O(n)$	$O(n)$	$O(n)$

1. Doubly linked list and position of k known
2. Position for insertion known

Figure 10.14: Comparison of various structures

EXERCISES

1. (a) Convince yourself that Figures 10.12 and 10.13 together with the cases for the symmetric rotations RR and RL takes care of all the possible situations that may arise when a height-balanced binary tree becomes unbalanced as a result of an insertion. Alternately, come up with an example that is not covered by any of the cases in this figure.
- (b) Draw the transformations for the rotation types RR and RL.

2. Show that the LR rotation of Figure 10.13 is equivalent to an RR rotation followed by an LL rotation and that an RL rotation is equivalent to an LL rotation followed by an RR rotation.
3. Prove by induction that the minimum number of nodes in an AVL tree of height h is $N_h = F_{h+2} - 1$, $h \geq 0$.
4. Complete `Avl::insert` (Program 10.3) by filling in the code needed to rebalance the tree in case of a right imbalance.
5. Start with an empty AVL tree and perform the following sequence of insertions: DECEMBER, JANUARY, APRIL, MARCH, JULY, AUGUST, OCTOBER, FEBRUARY, NOVEMBER, MAY, JUNE. Use the strategy of `Avl::insert` to perform each insert. Draw the AVL tree following each insertion and state the rotation type (if any) for each insert.
6. Assume that each node in an AVL tree has the data member `size`. For any node, n , $n \rightarrow \text{size}$ is the number of nodes in its left subtree plus one. Write a C++ function `Avl::Find(k)` to locate the k th smallest key in the tree. Show that this can be done in $O(\log n)$ time if there are n nodes in the tree.
7. Rewrite `Avl::insert` with the added assumption that each node has an `size` data member as in Exercise 6. Show that the insertion time remains $O(\log n)$.
8. Write a C++ function to list the elements of an AVL tree in ascending order of key. Show that this can be done in $O(n)$ time if the tree has n nodes.
9. Write an algorithm to delete the element with key k from an AVL tree. The resulting tree should be restructured if necessary. Show that the time required for this is $O(\log n)$ when there are n nodes in the tree. [Hint: If k is not in a leaf, then replace k by the largest value in its left subtree or the smallest value in its right subtree. Continue until the deletion propagates to a leaf. Deletion from a leaf can be handled using the reverse of the transformations used for insertion.]
10. Do Exercise 9 for the case when each node has an `size` data member and the k th smallest key is to be deleted.
11. Complete Figure 10.14 by adding a column for hashing.
12. For a fixed k , $k \geq 1$, we define a height-balanced tree $HB(k)$ as below:

Definition: An empty binary tree is an $HB(k)$ tree. If T is a nonempty binary tree with T_L and T_R as its left and right subtrees, then T is $HB(k)$ if (a) T_L and T_R are $HB(k)$ and (b) $|h_L - h_R| \leq k$, where h_L and h_R are the heights of T_L and T_R , respectively. \square

- (a) Obtain the rebalancing transformations for $HB(2)$.
- (b) Write an insertion algorithm for $HB(2)$ trees.

10.3 RED-BLACK TREES

10.3.1 Definition

A *red-black tree* is a binary search tree in which every node is colored either red or black. The remaining properties satisfied by a red-black tree are best stated in terms of the corresponding extended binary tree. Recall, from Section 9.2, that we obtain an extended binary tree from a regular binary tree by replacing every null pointer with an external node. The additional properties are

- RB1.** The root and all external nodes are colored black.
- RB2.** No root-to-external-node path has two consecutive red nodes.
- RB3.** All root-to-external-node paths have the same number of black nodes.

An equivalent definition arises from assigning colors to the pointers between a node and its children. The pointer from a parent to a black child is black and to a red child is red. Additionally,

- RB1'.** Pointers from an internal node to an external node are black.
- RB2'.** No root-to-external-node path has two consecutive red pointers.
- RB3'.** All root-to-external-node paths have the same number of black pointers.

Notice that if we know the pointer colors, we can deduce the node colors and vice versa. In the red-black tree of Figure 10.15, the external nodes are shaded squares, black nodes are shaded circles, red nodes are unshaded circles, black pointers are thick lines, and red pointers are thin lines. Notice that every path from the root to an external node has exactly two black pointers and three black nodes (including the root and the external node); no such path has two consecutive red nodes or pointers.

Let the *rank* of a node in a red-black tree be the number of black pointers (equivalently the number of black nodes minus 1) on any path from the node to any external node in its subtree. So the rank of an external node is 0. The rank of the root of Figure 10.15 is 2, that of its left child is 2, and of its right child is 1.

Lemma 10.1: Let the length of a root-to-external-node path be the number of pointers on the path. If P and Q are two root-to-external-node paths in a red-black tree, then $\text{length}(P) \leq 2\text{length}(Q)$.

Proof: Consider any red-black tree. Suppose that the rank of the root is r . From RB1' the last pointer on each root-to-external-node path is black. From RB2' no

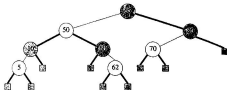


Figure 10.15: A red-black tree

such path has two consecutive red pointers. So each red pointer is followed by a black pointer. As a result, each root-to-external-node path has between r and $2r$ pointers, so $\text{length}(P) \leq 2\text{length}(Q)$. To see that the upper bound is possible, consider the red-black tree of Figure 10.15. The path from the root to the left child of 5 has length 4, while that to the right child of 80 has length 2. \square

Lemma 10.2: Let h be the height of a red-black tree (excluding the external nodes), let n be the number of internal nodes in the tree, and let r be the rank of the root.

- (a) $h \leq 2r$
- (b) $n \geq 2^r - 1$
- (c) $h \leq 2\log_2(n+1)$

Proof: From the proof of Lemma 10.1, we know that no root-to-external-node path has length $> 2r$, so $h \leq 2r$. (The height of the red-black tree of Figure 10.15 with external nodes removed is $2r = 4$.)

Since the rank of the root is r , there are no external nodes at levels 1 through r , so there are $2^r - 1$ internal nodes at these levels. Consequently, the total number of internal nodes is at least this much. (In the red-black tree of Figure 10.15, levels 1 and 2 have $3 = 2^2 - 1$ internal nodes. There are additional internal nodes at levels 3 and 4.)

From (b) it follows that $r \leq \log_2(n+1)$. This inequality together with (a)

yields (c). \square

Since the height of a red-black tree is at most $2\log_2(n+1)$, search, insert, and delete algorithms that work in $O(h)$ time have complexity $O(\log n)$.

Notice that the worst-case height of a red-black tree is more than the worst-case height (approximately $1.44\log_2(n+2)$) of an AVL tree with the same number of (internal) nodes.

10.3.2 Representation of a Red-Black Tree

Although it is convenient to include external nodes when defining red-black trees, in an implementation null pointers, rather than physical nodes, represent external nodes. Further, since pointer and node colors are closely related, with each node we need to store only its color or the color of the two pointers to its children. Node colors require just one additional bit per node, while pointer colors require two. Since both schemes require almost the same amount of space, we may choose between them on the basis of actual run times of the resulting red-black tree algorithms.

In our discussion of the insert and delete operations, we will explicitly state the needed color changes only for the nodes. The corresponding pointer color changes may be inferred.

10.3.3 Searching a Red-Black Tree

We can search a red-black tree with the code we used to search an ordinary binary search tree (Program 5.19). This code has complexity $O(h)$, which is $O(\log n)$ for a red-black tree. Since we use the same code to search ordinary binary search trees, AVL trees, and red-black trees and since the worst-case height of an AVL tree is least, we expect AVL trees to show the best worst-case performance in applications where search is the dominant operation.

10.3.4 Inserting into a Red-Black Tree

Elements may be inserted using the strategy used for ordinary binary trees (Program 5.21). When the new node is attached to the red-black tree, we need to assign the node a color. If the tree was empty before the insertion, then the new node is the root and must be colored black (see property RB1). Suppose the tree was not empty prior to the insertion. If the new node is given the color black,

then we will have an extra black node on paths from the root to the external nodes that are children of the new node. On the other hand, if the new node is assigned the color red, then we might have two consecutive red nodes. Making the new node black is guaranteed to cause a violation of property RB1, while making the new node red may or may not violate property RB2. We will make the new node red.

If making the new node red causes a violation of property RB2, we will say that the tree has become imbalanced. The nature of the imbalance is classified by examining the new node a , its parent pa , and the grandparent ga of a . Observe that since property RB2 has been violated, we have two consecutive red nodes. One of these red nodes is a , and the other must be its parent; therefore, pa exists. Since pa is red, it cannot be the root (as the root is black by property RB1); a must have a grandparent ga , which must be black (property RB2). When pa is the left child of ga , a is the left child of pa and the other child of ga is black (this case includes the case when the other child of ga is an external node); the imbalance is of type LLb. The other imbalance types are LLr (pa is the left child of ga , a is the left child of pa , the other child of ga is red), LRb (ga is the left child of ga , a is the right child of pa , the other child of ga is black), LRr, RRLb, RRLr, RLb, and RLr.

Imbalances of the type XYr (X and Y may be L or R) are handled by changing colors, while those of type XYb require a rotation. When we change a color, the RB2 violation may propagate two levels up the tree. In this case we will need to reclassify at the new level, with the new a being the former pa , and apply the transformations again. When a rotation is done, the RB2 violation is taken care of and no further work is needed.

Figure 10.16 shows the color changes performed for LLr and LLr imbalances; these color changes are identical. Black nodes are shaded, while red ones are not. In Figure 10.16(a), for example, ga is black, while pa and a are red; the pointers from ga to its left and right children are red; ga_L is the right subtree of ga ; and pa_R is the right subtree of pa . Both LLr and LRr color changes require us to change the color of pa and of the right child of ga from red to black. Additionally, we change the color of ga from black to red provided ga is not the root. Since this color change is not done when ga is the root, the number of black nodes on all root-to-external-node paths increases by 1 when ga is the root of the red-black tree.

If changing the color of ga to red causes an imbalance, ga becomes the new a node, its parent becomes the new pa , its grandparent becomes the new ga , and we continue to rebalance. If ga is the root or if the color change does not cause an RB2 violation at ga , we are done.

Figure 10.17 shows the rotations performed to handle LLb and LRb imbalances. In Figures 10.17(a) and (b), a is the root of pa_L . Notice the similarity between these rotations and the LL (refer to Figure 10.12) and LR (refer to



(a) LLr imbalance



(b) After LLr color change



(c) LRr imbalance



(d) After LRr color change

Figure 10.16: LLr and LRr color changes

Figure 10.13) rotations used to handle an imbalance following an insertion in an AVL tree. The pointer changes are the same. In the case of an LLr rotation, for example, in addition to pointer changes we need to change the color of gu from black to red and of pu from red to black.

In examining the node (or pointer) colors after the rotations of Figure 10.17, we see that the number of black nodes (or pointers) on all root-to-external-node paths is unchanged. Further, the root of the involved subtree (gu before the rotation and pu after) is black following the rotation; therefore, two consecutive red nodes cannot exist on the path from the tree root to the new pu .

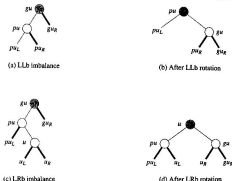


Figure 10.17: LLb and LRb rotations for red-black insertion

Consequently, no additional rebalancing work is to be done. A single rotation (preceded possibly by $O(\log n)$ color changes) suffices to restore balance following an insertion!

Example 10.4: Consider the red-black tree of Figure 10.18(a). External nodes are shown for convenience. In an actual implementation, the shown black pointers to external nodes are simply null pointers and external nodes are not represented. Notice that all root-to-external-node paths have three black nodes (including the external node) and two black pointers.

To insert 70 into this red-black tree, we use the algorithm of Program 10.4. The new node is added as the left child of 80. Since the insertion is done into a nonempty tree, the new node is assigned the color red. So the pointer to it from its parent (80) is also red. This insertion does not result in a violation of property

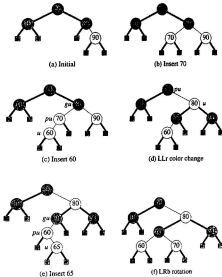


Figure 10.18: Insertion into a red-black tree (continued on next page)

RB2, and no remedial action is necessary. Notice that the number of black pointers on all root-to-external-node paths is the same as before the insertion.

Next insert 60 into the tree of Figure 10.18(b). The algorithm of Program 5.23 attaches a new node as the left child of 70, as is shown in Figure 10.18(c). The new node is red, and the pointer to it is also red. The new node is the w node, its parent (70) is pu , and its grandparent (80) is gx . Since pu and w are red, we have an imbalance. This imbalance is classified as an LLr imbalance (as pu is the left child of gx , w is the left child of pu , and the other child of gx is red). When the LLr color change of Figure 10.16(a) and (b) is performed, we get the tree of Figure 10.18(d). Now w , pu , and gx are each moved two levels up the tree. The node with 80 is the new w node, the root becomes pu , and gx is NULL. Since there is no gx node, we cannot have an RB2 imbalance at this location and we are done. All root-to-external-node paths have exactly two black pointers.

Now insert 65 into the tree of Figure 10.18(d). The result appears in Figure 10.18(e). The new node is the w node. Its parent and grandparent are, respectively, the pu and gx nodes. We have an LRb imbalance that requires us to perform the rotation of Figures 10.17(c) and (d). The result is the tree of Figure 10.18(f).

Finally, insert 62 to obtain the tree of Figure 10.18(g). We have an LRr imbalance that requires a color change. The resulting tree and the new w , pu , and gx nodes appear in Figure 10.18(h). The color change just performed has caused an RLb imbalance two levels up, so we now need to perform an RLb rotation. The rotation results in the tree of Figure 10.18(i). Following a rotation, no further work is needed, and we are done. \square

10.3.5 Deletion from a Red-Black Tree

The development of the deletion transformations is left as an exercise.

10.3.6 Joining Red-Black Trees

In Section 5.7.5, we defined the following operations on binary search trees: *ThreeWayJoin*, *TwoWayJoin*, and *Split*. Each of these can be performed in logarithmic time on red-black trees. The operation *ThreeWayJoin* (A , x , B) can be performed as follows. A two-way join may be done in a similar fashion.

- Case 1:** If A and B have the same rank, then let C be constructed by creating a new root with pair x , *leftChild* A , and *rightChild* B . Both links are made black. The rank of C is one more than the ranks of A and B .
- Case 2:** If $\text{rank}(A) > \text{rank}(B)$, then follow *rightChild* pointers from A to the first node P that has rank equal to $\text{rank}(B)$. Properties RB1 to RB3

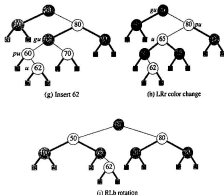


Figure 10.18: Insertion into a red-black tree

guarantee the existence of such a node. Let $p(Y)$ be the parent of Y . From the definition of Y , it follows that $\text{rank}(p(Y)) = \text{rank}(Y) + 1$. Hence, the pointer from $p(Y)$ to Y is a black pointer. Create a new node, Z , with pair x , $\text{leftChild } Y$ (i.e., node Y and its subtrees become the left subtree of Z) and $\text{rightChild } B$. Z is made the right child of $p(Y)$, and the link from $p(Y)$ to Z has color red. The links from Z to its children are made black. Note that this transformation does not change the number of black pointers on any root-to-external-node path. However, it may cause the path from the root to Z to contain two consecutive red pointers. If this happens, then the transformations used to

handle this in a bottom-up insertion are performed. These transformations may increase the rank of the tree by one.

Case 3: The case $\text{rank}(A) < \text{rank}(B)$ is similar to Case 2.

Analysis of ThreeWayJoin: The correctness of the function just described is easily established. Case 1 takes $O(1)$ time; each of the remaining two cases takes $O(\text{rank}(A) - \text{rank}(B) + 1)$ time under the assumption that the rank of each red-black tree is known prior to computing the join. Hence, a three-way join can be done in $O(\log n)$ time, where n is the number of nodes in the two trees being joined. A two-way join can be performed in a similar manner. Note that there is no need to add parent data members to the nodes to perform a join, as the needed parents can be saved on a stack as we move from the root to the node Z . \square

10.3.7 Splitting a Red-Black Tree

We now turn our attention to the split operation. Assume for simplicity that the splitting key, i , is actually present in the red-black tree A . Under this assumption, the split operation $A.\text{Split}(i, B, x, C)$ can be performed as in Program 10.5.

Step 1: Search A for the node P that contains the element with key i . Copy this element to the reference parameter x . Initialize B and C to be the left and right subtrees of P , respectively.

Step 2:

```

for ( $Q = \text{parent}(P)$ ;  $Q \neq P$ ;  $Q = \text{parent}(Q)$ )
    if ( $P == Q \rightarrow \text{leftChild}$ )
         $C.\text{ThreeWayJoin}(C, Q \rightarrow \text{data}, Q \rightarrow \text{rightChild})$ 
    else  $B.\text{ThreeWayJoin}(Q \rightarrow \text{leftChild}, Q \rightarrow \text{data}, B)$ ;

```

Program 10.5: Splitting a red-black tree

We first locate the splitting element, x , in the red-black tree. Let P be the node that contains this element. The left subtree of P contains elements with key less than i . B is initialized to be this subtree. All elements in the right subtree of P have a key larger than i , and C is initialized to be this subtree. In Step 2, we trace the path from P to the root of the red-black tree A . During this traceback, two kinds of subtrees are encountered. One of these contains elements with keys that are larger than i as well as all keys in C . This happens when the traceback moves from a left child of a node to its parent. The other kind of subtree

contains elements with keys that are smaller than i , as well as smaller than all keys in B . This happens when we move from a right child to its parent. In the former case a three-way join with C is performed, and in the latter a three-way join with B is performed. One may verify that the two-step procedure outlined here does indeed implement the split operation for the case when i is the key of an element in the tree A . It is easily extended to handle the case when the tree A contains no element with key i .

Analysis of split: Call a node *red* if the pointer to it from its parent is red. The root and all nodes that have a black pointer from their parent are black. Let $r(X)$ be the rank of node X in the unsplit tree. First, we shall show that during a split, if P is a black node in the unsplit tree, and $Q \neq 0$, then

$$r(Q) \geq \max\{r(B), r(C)\}$$

where P , Q , B , and C are as defined at the start of an iteration of the for loop in Step 2.

From the definition of rank, the inequality holds at the start of the first iteration of the for loop regardless of the color of P . If P is red initially, then its parent, Q , exists and is black. Let q^+ be the parent of Q . If $q^+ = 0$, then there is no Q at which the inequality is violated. So, assume $q^+ \neq 0$. From the definition of rank and the fact that Q is black, it follows that $r(q^+) = r(Q) + 1$. Let B^+ and C^+ be the trees B and C following the three-way join of Step 2. Since $r(B^+) \leq r(B) + 1$ and $r(C^+) \leq r(C) + 1$, $r(q^+) = r(Q) + 1 \geq \max\{r(B), r(C)\} + 1 \geq \max\{r(B^+), r(C^+)\}$. So, the inequality holds the first time Q points to a node with a black child P (i.e., at the start of the second iteration of the for loop, when $Q = q^+$).

Having established the induction base, we can proceed to show that the inequality holds at all subsequent iterations when Q points to a node with a black child P . Suppose Q is currently pointing to a node q with a black child $P = p$. Assume that the inequality holds. We shall show that it will hold again the next time Q is at a node with a black P . For there to be such a next time, the parent q^+ of q must exist. If q is black, the proof is similar to that provided for a black Q and a red P in the induction base.

If q is red, then q^+ is black. Further, for there to be a next time when Q is at a node with a black P , q must have a grandparent q^{++} , as when Q moves to q^+ and P to q , $Q = q^+$ has a red child $P = q$. Let B^+ and C^+ represent the B and C trees following the iteration that begins with $P = p$ and $Q = q$. Similarly, let B^{++} and C^{++} represent these trees following the iteration that begins with $P = q$ and $Q = q^+$.

Suppose that C is joined with q and its right subtree R to create C^+ . If $r(C) = r(R)$, then $r(C^+) = r(C) + 1$, and C^+ has two black children (recall that when the rank increases by one, the root has two black children). If $C^+ = C$, then $B =$

B' is combined with q' and its left subtree L' to form B'' . Since $r(L') \leq r(q')$, $r(B'') \leq \max\{r(B'), r(L')\} + 1$, and $r(q'') = r(q') + 1 = r(q) + 1$, $r(B'') \leq r(q'')$. Also, $r(C'') = r(C') = r(C) + 1 \leq r(q) + 1 \leq r(q'')$. So, the inequality holds when $Q = q''$. If $C'' \neq C'$, then C'' is combined with q' and its right subtree R' to form C'' . If $r(R') \geq r(C')$, then $r(C'') \leq r(R') + 1 \leq r(q') + 1 = r(q'')$. If $r(R') < r(C')$, then $r(C'') = r(C')$, as C' has two black children, and the join of C' , q' , and R' does not increase the rank. Once again, $r(C'') \leq r(q'')$, and the inequality holds when $Q = q''$.

If $r(C) > r(R)$ and $r(C') = r(C)$, then $r(q'') = r(q) + 1 \geq \max\{r(B'), r(C')\} + 1 \geq \max\{r(B''), r(C'')\}$. If $r(C') = r(C) + 1$, then C' has two black children, and $r(C'') \leq r(q) + 1 = r(q'')$. Also, $r(B'') \leq r(q'')$. So, the inequality holds when $Q = q''$. The case $r(C) < r(R)$ is similar.

The case when B is joined with q and its left subtree L is symmetric.

Using the rank inequality just established, we can show that whenever Q points to a node with a black child, the total work done in Step 2 of the splitting algorithm from initiation to the time Q reaches this node is $O(r(B) + r(C) + r(Q))$. Here, B and C are, respectively, the current red-black trees with values smaller and larger than the splitting value. Since $r(Q) \geq \max\{r(B), r(C)\}$, the total work done in Step 2 is $O(r(Q))$. From this, it follows that the time required to perform a split is $O(\log n)$, where n is the number of nodes in the tree to be split. \square

EXERCISES

1. Start with an empty red-black tree and insert the following keys in the given order: 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1. Draw figures similar to Figure 10.18 depicting your tree immediately after each insertion and following the rebalancing rotation or color change (if any). Label all nodes with their color and identify the rotation type (if any) that is done.
2. Do Exercise 1 using the insert sequence: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15.
3. Do Exercise 1 using the insert key sequence: 20, 10, 5, 30, 40, 57, 3, 3, 4, 35, 25, 18, 22, 21.
4. Do Exercise 1 using the insert key sequence: 40, 50, 70, 30, 42, 15, 20, 25, 27, 26, 60, 55.
5. Draw the RBr and RLr color changes that correspond to the LLr and LRr changes of Figure 10.16.
6. Draw the RRb and RLb rotations that correspond to the LLb and LRb changes of Figure 10.17.

7. Let T be a red-black tree with rank r . Write a function to compute the rank of each node in the tree. The time complexity of your function should be linear in the number of nodes in the tree. Show that this is the case.
8. Compare the worst-case height of a red-black tree with n nodes and that of an AVL tree with the same number of nodes.
9. Develop the deletion transformations for a red-black tree. Show that a deletion from a red-black tree requires at most one rotation.
10. (a) Use the strategy described in this section to obtain a C++ function to compute a three-way join. Assume the existence of a function *rebalance*(X) that performs the necessary transformations if the tree pointer to node X is the second of two consecutive red pointers. The complexity of this function may be assumed to be $O(\text{level}(X))$.
 (b) Prove the correctness of your function.
 (c) What is the time complexity of your function?
11. Obtain a function to perform a two-way join on red-black trees. You may assume the existence of functions to search, insert, delete, and perform a three-way join. What is the time complexity of your two-way join function?
12. Use the strategy suggested in Program 10.5 to obtain a C++ function to perform the split operation in a red-black tree T . The complexity of your algorithm must be $O(\text{height}(T))$. Your function must work when the splitting key i is present in T and when it is not present in T .
13. Complete the complexity proof for the split operation by showing that whenever Q has a black child, the total work done in Step 2 of the splitting algorithm from initiation to the time that Q reaches the current node is $O(r(Q))$.
14. Program the search, insert, and delete operations for AVL trees and red-black trees.
 - (a) Test the correctness of your functions.
 - (b) Generate a random sequence of n inserts of distinct values. Use this sequence to initialize each of the data structures. Next, generate a random sequence of searches, inserts, and deletes. In this sequence, the probability of a search should be 0.5, that of an insert 0.25, and that of a delete 0.25. The sequence length is m . Measure the time needed to perform the m operations in the sequence using each of the above data structures.
 - (c) Do part (b) for $n = 100, 1000, 10,000$, and $100,000$ and $m = n, 2n$, and $4n$.
 - (d) What can you say about the relative performance of these data

structures?

10.4 SPLAY TREES

We have studied balanced search trees that allow one to perform operations such as search, insert, delete, join, and split in $O(\log n)$ worst-case time per operation. In the case of priority queues, we saw that if we are interested in amortized complexity rather than worst-case complexity, simpler structures can be used. This is also true for search trees. Using a splay tree, we can perform the operations in $O(\log n)$ amortized time per operation. In this section, we develop two varieties of splay trees—bottom up and top down. Although the amortized complexity of each operation is $O(\log n)$ for both varieties, experiments indicate that top-down splay trees are faster than bottom-up splay trees by a constant factor.

10.4.1 Bottom-Up Splay Trees

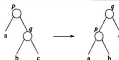
A *bottom-up splay tree* is a binary search tree in which each search, insert, delete, and join operation is performed in the same way as in an ordinary binary search tree (see Chapter 5) except that each of these operations is followed by a *splay*. In a split, however, we *first* perform a splay. This makes the split very easy to perform. A splay consists of a sequence of rotations. For simplicity, we assume that each of the operations is always successful. A failure can be modeled as a different successful operation. For example, an unsuccessful search may be modeled as a search for the element in the last node encountered in the unsuccessful search, and an unsuccessful insert may be modeled as a successful search. With this assumption, the start node for a splay is obtained as follows:

- (1) *Search*: The splay starts at the node containing the element being sought.
- (2) *Insert*: The start node for the splay is the newly inserted node.
- (3) *Delete*: The parent of the physically deleted node is used as the start node for the splay. If this node is the root, then no splay is done.
- (4) *ThreeWayJoin*: No splay is done.
- (5) *Split*: Suppose that we are splitting with respect to the key i and that key i is actually present in the tree. We first perform a splay at the node that contains i and then split the tree. As we shall see, splitting following a splay is very simple.

Splay rotations are performed along the path from the start node to the root of the binary search tree. These rotations are similar to those performed for AVL

trees, and red-black trees. Let q be the node at which the splay is being performed. Initially, q is the node at which the splay starts. The following steps define a splay:

- (1) If q is either 0 or the root, then the splay terminates.
- (2) If q has a parent, p , but no grandparents, then the rotation of Figure 10.19 is performed, and the splay terminates.



a, b , and c are subtrees

Figure 10.19: Rotation when q is a right child and has no grandparent

- (3) If q has a parent, p , and a grandparent, gp , then the rotation is classified as LL (p is the left child of gp , and q is the left child of p), LR (p is the left child of gp , and q is the right child of p), RR, or RL. The RR and RL rotations are shown in Figure 10.20. LL and LR rotations are symmetric to these. The splay is repeated at the new location of q .

Notice that all rotations move q up the tree and that following a splay, q becomes the new root of the search tree. As a result, splitting the tree with respect to a key, k , is done simply by performing a splay at k and then splitting at the root. Figure 10.21 shows a binary search tree before, during, and after a splay at the shaded node.

In the case of Fibonacci heaps, we obtained the amortized complexity of an operation by using an explicit cross-charging scheme. The analysis for splay trees will use a potential technique. Let P_0 be the initial potential of the search tree, and let P_i be its potential following the i th operation in a sequence of n operations. The amortized time for the i th operation is defined to be

$$(\text{actual time for the } i\text{th operation}) + P_i - P_{i-1}$$

That is, the amortized time is the actual time plus the change in the potential.

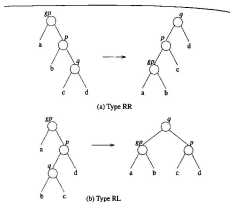


Figure 10.20: RR and RL rotations

Rearranging terms, we see that the actual time for the i th operation is

$$(\text{amortized time for the } i\text{th operation}) + P_{i+1} - P_i$$

Hence, the actual time needed to perform the m operations in the sequence is

$$\sum_i (\text{amortized time for the } i\text{th operation}) + P_m - P_0$$

Since each operation other than a join involves a splay whose actual complexity is of the same order as that of the whole operation, and since each join takes $O(1)$ time, it is sufficient to consider only the time spent performing splays.

Each splay consists of several rotations. We shall assign to each rotation a fixed cost of one unit. The choice of a potential function is rather arbitrary. The

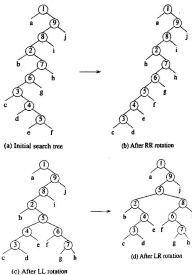
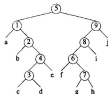


Figure 10.21: Rotations in a splay beginning at shaded node (continued on next page)



(c) After RL rotation

Figure 10.21: Rotations in a splay beginning at the shaded node

objective is to use one that results in as small a bound on the time complexity as is possible. We now define the potential function we shall use. Let the size, $s(i)$, of the subtree with root i be the total number of nodes in it. The rank, $r(i)$, of node i is equal to $\lfloor \log_2 s(i) \rfloor$. The potential of the tree is $\sum_i r(i)$. The potential of an empty tree is defined to be zero.

Suppose that in the tree of Figure 10.21(a), the subtrees a, b, \dots, j are all empty. Then $(s(1), \dots, s(9)) = (9, 6, 3, 2, 1, 4, 5, 7, 8)$; $r(3) = r(4) = 1$; $r(5) = 0$; and $r(9) = 3$. In Lemma 10.3 we use r and r' , respectively, to denote the rank of a node before and after a rotation.

Lemma 10.3: Consider a binary search tree that has n elements/nodes. The amortized cost of a splay operation that begins at node q is at most $3(\lfloor \log_2 n \rfloor - r(q)) + 1$.

Proof: Consider the three steps in the definition of a splay:

- (1) In this case, q either is 0 or the root. This step does not alter the potential of the tree, so its amortized and actual costs are the same. This cost is 1.
- (2) In this step, the rotation of Figure 10.19 (or the symmetric rotation for the case when q is the left child of p) is performed. Since only the ranks of p and q are affected, the potential change, ΔP , is $r'(p) + r'(q) - r(p) - r(q)$. Further, since $r'(p) \leq r(p)$, $\Delta P \leq r'(q) - r(q)$. The amortized cost of this step (actual cost plus potential change) is, therefore, no more than

$$r'(q) = r(q) + 1.$$

- (3) In this step only the ranks of q , p , and gp change. So, $\Delta P = r'(q) + r'(p) + r'(gp) - r(q) - r(p) - r(gp)$. Since, $r(gp) = r'(q)$,

$$\Delta P = r'(p) + r'(gp) - r(q) - r(p) \quad (1)$$

Consider an RR rotation. From Figure 10.20(a), we see that $r'(p) \leq r'(q)$, $r'(gp) \leq r'(q)$, and $r(q) \leq r(p)$. So, $\Delta P \leq 2(r'(q) - r(q))$. If $r'(q) > r(q)$, $\Delta P \leq 2(r'(q) - r(q)) - 1$. If $r'(q) = r(q)$, then $r'(q) = r(q) = r(p) = r(gp)$. Also, $r'(q) > r(q) + r'(gp)$. Consequently, $r'(gp) < r'(q)$. To see this, note that if $r'(gp) = r'(q)$, then $r'(q) > 2^{h(q)} + 2^{h(gp)} = 2^{r(q)+1}$, which violates the definition of rank. Hence, from (1), $\Delta P \leq 2(r'(q) - r(q)) - 1 = 3(r'(q) - r(q)) - 1$. So, the amortized cost of an RR rotation is at most $1 + 3(r'(q) - r(q)) - 1 = 3(r'(q) - r(q))$.

This bound may be obtained for LL, LR, and RL rotations in a similar way.

The lemma now follows by observing that Steps 1 and 2 are mutually exclusive and can occur at most once. Step 3 occurs zero or more times. Summing up over the amortized cost of a single occurrence of Steps 1 or 2 and all occurrences of Step 3, we obtain the bound of the lemma. \square

Theorem 10.1: The total time needed for a sequence of m search, insert, delete, join, and split operations performed on a collection of initially empty splay trees is $O(m \log n)$, where $n, m > 0$, is the number of inserts in the sequence.

Proof: Since none of the splay trees has more than n nodes, no node has rank more than $\lfloor \log_2 n \rfloor$. A search (excluding the splay) does not change the rank of any node and hence does not affect the potential of the splay tree involved. An insert (excluding the splay) increases, by one, the size of every node on the path from the root to the newly inserted node. This causes the ranks of the nodes with size $2^h - 1$ to change. There are at most $\lfloor \log_2 n \rfloor + 1$ such nodes on any insert path. So, each insert (excluding the splay) increases the potential by at most this much. Each join increases the total potential of all the splay trees by at most $\lfloor \log_2 n \rfloor$. Deletions do not increase the potential of the involved splay tree except for any increase that results from the splay step. The split operation (excluding the splay step) reduces the overall potential by an amount equal to the rank of the tree just before the split (but after the splay that precedes it). So, the potential increase, P_I , attributable to the m operations (exclusive of that attributable to the splay steps of the operations) is $O(m \log n)$.

From our definition of the amortized cost of a splay operation, it follows that the time for the sequence of operations is the sum of the amortized costs of the splays, the potential change $P_F - P_m$, and P_I . From Lemma 10.3, it follows

598 Efficient Binary Search Trees

that the sum of the amortized costs is $O(m \log n)$. The initial potential, P_0 , is 0, and the final potential, P_m , is ≥ 0 . So, the total time is $O(m \log n)$. \square

10.4.2 Top-Down Splay Trees

As in the case of a bottom-up splay tree, a *ThreeWayJoin* is implemented the same in a top-down splay tree as in Section 5.7.5. For the remaining operations, let the splay node be as defined for a bottom-up splay tree. For each operation, we follow a path from the root to the splay node as in Section 5.7.5. However, as we follow this path, we partition the binary search tree into three components—a binary search tree *small* of elements whose key is less than that of the element in the splay node, a binary search tree *big* of elements whose key is greater than that of the element in the splay node, and the splay node. Notice that in this downward traversal of the path from the root to the splay node, we do not actually know which node is the splay node until we get to it. So, the downward traversal is done using the key k associated with the operation that is being performed.

For the partitioning, we begin with two empty binary search trees *small* and *big*. It is convenient to give these trees a header node that is deleted at the end. Let s and b , respectively, be initialized to the header nodes of *small* and *big*. The downward traversal to the splay node begins at the root. Let x denote the node we currently are at. We begin with x being the tree root. There are 7 cases to consider:

Case 0: x is the splay node.

Terminate the partitioning process.

Case L: *The splay node is the left child of x .*

In this case, x and its right subtree contain keys that are greater than that in the splay node. So, we make x the left child of b ($b \rightarrow \text{leftChild} = x$) and set $b = x$ and $x = x \rightarrow \text{leftChild}$. Notice that this automatically places the right subtree of x into *big*. Figure 10.22 shows a schematic for this case.

Case R: *The splay node is the right child of x .*

This case is symmetric to Case L. Now, x and its left subtree contain keys that are less than that in the splay node. So, we make x the right child of s ($s \rightarrow \text{rightChild} = x$) and set $s = x$ and $x = x \rightarrow \text{rightChild}$. Notice that this automatically places the left subtree of x into *small*.

Case LR: *The splay node is in the right subtree of the left child of x .*

This case is handled as Case L followed by Case R.

Case RL: *The splay node is in the left subtree of the right child of x .*

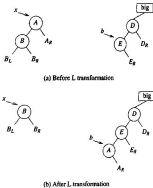


Figure 10.22: Case L for a top-down splay tree

This case is handled as Case R followed by Case L.

Case LL: *The splay node is in the left subtree of the left child of x .*

This case is *not* handled as two applications of Case L. Instead we perform an LL rotation around x . Figure 10.23 shows a schematic for this case. The shown transformation is accomplished by the following code fragment:

```

b → leftChild = x → leftChild;
b = b → leftChild;
x → leftChild = b → rightChild;
b → rightChild = x;

```

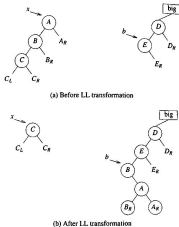



Figure 18.13: Case LL for a top-down splay tree

$$x = b \rightarrow \text{leftChild};$$

Case RR: The splay node is in the right subtree of the right child of x .
This case is symmetric to Case LL.

The above transformations are applied repeatedly until terminated by an application of Case 0. Upon termination, x is the splay node. Now, the left

subtree of x is made the right subtree of s and the right subtree of x is made the left subtree of b . Finally, the header nodes of the small and big trees are deleted.

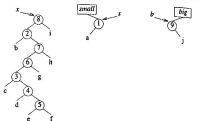
In case we were performing a split operation and x contains the split key, we return *small*, $x \rightarrow \text{data}$, and *big* as the result of the split. For a search, insert and delete, we make *small* and *big*, respectively, the left and right subtrees of s and the tree rooted at x is the new binary search tree (we assume that in the downward quest for the splay node, the remaining tasks associated with the search, insert and delete operations have been done).

Example 10.5: Suppose we are searching for the key 5 in the top-down splay tree of Figure 10.21(a). Although we don't know this at this time, the splay node is the shaded node. The path from the root to the splay node is determined by comparing the search key 5 with the key in the current node. We start with the current node pointer x at the root and two empty splay trees—*small* and *big*. These empty splay trees have a header node. The variables s and b , respectively, point to these header nodes. Since the splay node is in the left subtree of the right child of x , an RL transformation is called for. The search tree as well as the trees *small* and *big* following the RL transformation are shown in Figure 10.24(a).

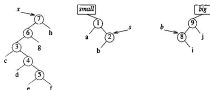
Now, since the splay node is in the right subtree of the left child of the new x , an LR transformation is made and we obtain the configuration of Figure 10.24(b). Next, we make an LL transformation (Figure 10.24(c)) and an RR transformation (Figure 10.24(d)). Now, x is at the splay node. The left subtree of x is made the right subtree of s and the right subtree of x is made the left subtree of b (Figure 10.24(e)). Finally, we delete the header nodes and make the small and big trees subtrees of s as shown in Figure 10.24(f). □

EXERCISES

1. Obtain figures corresponding to Figures 10.19 and 10.20 for the symmetric bottom-up splay tree rotations.
2. What is the maximum height of a bottom-up splay tree that is created as the result of n insertions made into an initially empty splay tree? Give an example of a sequence of inserts that results in a splay tree of this height.
3. Complete the proof of Lemma 10.3 by providing the proof for the case of an RL rotation. Note that the proofs for LL and LR rotations are similar to those for RR and RL rotations, respectively, as the rotations are symmetric.
4. Explain how a two-way join should be performed in a bottom-up splay tree so that the amortized cost of each splay tree operation remains $O(\log n)$.
5. Explain how a split with respect to key i is to be performed when key i is not present in the bottom-up splay tree. The amortized cost of each bottom-up splay tree operation should be $O(\log n)$.



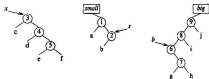
(a) After RL transformation



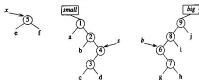
(b) After LR transformation

Figure 10.24: Example for top-down splay tree (continued on next page)

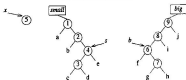
6. Implement the bottom-up splay tree data structure as a C++ class. Test all functions using your own test data.
7. [Sleator and Tarjan] Suppose we modify the definition of $s(i)$ used in connection with the complexity analysis of bottom-up splay trees. Let each node i have a positive weight $w(i)$. Let $s(i)$ be the sum of the weights of all



(c) After LL transformation

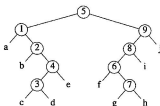


(d) After RR transformation



(e) After moving subtrees of splay node

Figure 10.24: Example for top-down splay tree (continued on next page)



(f) Final search tree

Figure 10.24: Example for top-down splay tree

nodes in the subtree with root i . The rank of this subtree is $\log_2 s(i)$.

- (a) Let t be the root of a splay tree. Show that the amortized cost of a splay that begins at node q is at most $3(r(t) - r(q)) + 1$, where r is the rank just before the splay.
- (b) Let S be a sequence of n inserts and m searches. Assume that each of the n inserts adds a new element to the splay tree and that all searches are successful. Let $p(i)$, $p(i) > 0$, be the number of times element i is sought. The $p(i)$'s satisfy the following equality:

$$\sum_{i=1}^n p(i) = m$$

Show that the total time spent on the m searches is

$$O(m + \sum_{i=1}^n p(i) \log(m/p(i)))$$

Note that since $\Omega(m + \sum_{i=1}^n p(i) \log(m/p(i)))$ is an information theoretic bound on the search time in a static search tree (the optimal binary search tree of Section 10.1 is an example of such a tree), bottom-up splay trees are optimal to within a constant factor for the representation of a static set of elements.

8. Obtain figures corresponding to Figures 10.22 and 10.23 for the top-down splay tree transformations R, RR, RL, and LR.
9. What is the maximum height of a top-down splay tree that is created as the result of n insertions made into an initially empty splay tree? Give an example of a sequence of inserts that results in a splay tree of this height.
10. Implement the top-down splay tree data structure as a C++ class. Test all functions using your own test data.

10.5 REFERENCES AND SELECTED READINGS

The $O(n^2)$ optimum binary search tree algorithm is from "Optimum binary search trees," by D. Knuth, *Acta Informatica*, 1:1, 1971, pp. 14–25. For a discussion of heuristics that obtain in $O(n \log n)$ time nearly optimal binary search trees, see "Nearly optimal binary search trees," by K. Mehlhorn, *Acta Informatica*, 5, 1975, pp. 287–295; and "Binary search trees and file organization," by J. Nievergelt, *ACM Computing Surveys*, 6:3, 1974, pp. 195–207.

The original paper on AVL trees by G. M. Adelson-Velskii and E. M. Landis appears in *Dokl. Acad. Nauk, SSR (Soviet Math)*, 3, 1962, pp. 1259–1263. Additional algorithms to manipulate AVL trees may be found in "Linear lists and priority queues as balanced binary trees," by C. Chase, Technical Report STAN-CS-72-259, Computer Science Dept., Stanford University, Palo Alto, CA, 1972; and *The Art of Computer Programming: Sorting and Searching* by D. Knuth, Addison-Wesley, Reading, MA, 1973 (Section 6.2.3).

Results of an empirical study of height-balanced trees appear in "Performance of height-balanced trees," by P. L. Kartin, S. H. Paller, R. E. Scroggs, and E. B. Koehler, *CACM*, 19:1, 1976, pp. 23–28.

Splay trees were invented by D. Sleator and R. Tarjan. Their paper "Self-adjusting binary search trees," *JACM*, 32:3, 1985, pp. 652–686, provides several other analyses of splay trees, as well as variants of the basic splaying technique discussed in the text. Our analysis is modeled after that in *Data Structures and Network Algorithms*, by R. Tarjan, SIAM Publications, Philadelphia, PA, 1983.

For more on binary search trees, see Chapters 10 through 14 of "Handbook of data structures and applications," edited by D. Mehta and S. Sahai, Chapman & Hall/CRC, Boca Raton, 2005.

Multiway Search Trees

11.1 *m*-WAY SEARCH TREES

11.1.1 Definition and Properties

Balanced binary search trees such as AVL and red-black trees allow us to search, insert, and delete in $O(\log n)$ time, where n is the number of elements. While this may seem to be a remarkable accomplishment, we can improve the performance of search structures by capitalizing on the exorbitant time it takes to make a memory access (whether to main memory or to disk) relative to the time it takes to perform an arithmetic or logic operation in a modern computer. An access to main memory typically takes approximately 100 times the time to do an arithmetic operation while an access to disk takes about 10,000 times the time for an arithmetic operation. Because of this significant mismatch between processor speed and memory access time, data is typically moved from main memory to cache (fast memory) in units of a cache-line size (of the order of 100 bytes) and from disk to main memory in units of a block (several kilo bytes). For uniformity with disks, we say that main memory is organized into blocks: the

size of each block being equal to that of a cache line. AVL and red-black trees are unable to take advantage of this large unit (i.e., block) in which data is moved from slow memory (main or disk) to faster memory (cache or main) since the node size is typically only a few bytes. Consider an AVL tree with 1,000,000 elements. Its height may be as much as $\lceil 1.44\log_2(n+2) \rceil$, which is 28. To search this tree for an element with a specified key, we must access those nodes that are on the search path from the root to the node that contains the desired element. This path may contain 28 nodes and if each of these 28 nodes lies in a different memory block, a total of 28 memory accesses and 28 compares are made in the worst case. Most of the search time is spent on memory access! To improve performance, we must reduce the number of memory accesses. Notice that if halving the number of memory accesses resulted in a doubling of the number of comparisons, we would still achieve a reduction in total search time. Since the number of memory accesses is closely tied to the height of the search tree, we must reduce tree height. To break the $\log_2(n+1)$ barrier on tree height resulting from the use of binary search trees, we must use search trees whose degree is more than 2. In practice, we use the largest degree for which the tree node fits into a block (whether cache line or disk block).

Definition: An *m*-way search tree is either empty or satisfies the following properties:

- (1) The root has at most *m* subtrees and has the following structure:

$$n, A_0, (E_1, A_1), (E_2, A_2), \dots, (E_n, A_n)$$

where the A_i , $0 \leq i \leq n$, are pointers to subtrees, and the E_i , $1 \leq i \leq n$, are elements. Each element E_i has a key $E_i.K$.

- (2) $E_i.K < E_{i+1}.K$, $1 \leq i < n$.
- (3) Let $E_0.K = -\infty$ and $E_{n+1}.K = \infty$. All keys in the subtree A_i are less than $E_{i+1}.K$ and greater than $E_i.K$, $0 \leq i \leq n$.
- (4) The subtrees A_i , $0 \leq i \leq n$, are also *m*-way search trees. \square

We may verify that binary search trees are two-way search trees. A three-way search tree is shown in Figure 11.1. For convenience, only keys are shown in this figure as well as in all remaining figures in this chapter.

In a tree of degree *m* and height *h*, the maximum number of nodes is

$$\sum_{0 \leq i \leq h-1} m^i = (m^h - 1)/(m - 1)$$

Since each node has at most *m* - 1 elements, the maximum number of elements in an *m*-way tree of height *h* is $m^h - 1$. For a binary tree with $h = 3$ this quantity

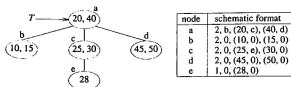


Figure 11.1: Example of a three-way search tree

is 7. For a 200-way tree with $h = 3$ we have $m^h - 1 = 8 \times 10^6 - 1$.

To achieve a performance close to that of the best m -way search trees for a given number of elements n , the search tree must be balanced. The particular varieties of balanced m -way search trees we shall consider here are known as B-trees and B⁺-trees.

11.1.2 Searching an m -Way Search Tree

Suppose we wish to search an m -way search tree for an element whose key is x . We begin at the root of the tree. Assume that this node has the structure given in the definition of an m -way search tree. For convenience, assume that $E_0.K = -\infty$ and $E_{n+1}.K = +\infty$. By searching the keys of the root, we determine i such that $E_i.K \leq x < E_{i+1}.K$. If $x = E_i.K$, then the search is complete. If $x \neq E_i.K$, then from the definition of an m -way search tree, it follows that if x is in the tree, it must be in subtree A_i . So, we move to the root of this subtree and proceed to search it. This process continues until either we find x or we have determined that x is not in the tree (the search leads us to an empty subtree). When the number of elements in the node being searched is small, a sequential search is used. When this number is large, a binary search may be used. A high-level description of the algorithm to search an m -way search tree is given in Program 11.1.

```

// Search an  $m$ -way search tree for an element with key  $x$ .
// Return the element if found. Return NULL otherwise.
 $E_0.K = -\text{MAXKEY}$ ;
for ( $*p = \text{root}$ ;  $p.K \neq A_0$ )
{
    Let  $p$  have the format  $A, A_0, (E_1, A_1), \dots, (E_n, A_n)$ ;
     $E_{n+1}.K = \text{MAXKEY}$ ;
    Determine  $i$  such that  $E_i.K \leq x < E_{i+1}.K$ ;
    if ( $x == E_i.K$ ) return  $E_i$ ;
}
//  $x$  is not in the tree
return NULL;

```

Program 11.1: Searching an m -way search tree

EXERCISES

1. Draw a sample 5-way search tree.
2. What is the minimum number of elements in an m -way search tree whose height is h ?
3. Write an algorithm to insert an element whose key is x into an m -way search tree. What is the complexity of your algorithm?
4. Write an algorithm to delete the element whose key is x from an m -way search tree. What is the complexity of your algorithm?

11.2 B-TREES

11.2.1 Definition and Properties

In defining a B-tree, it is convenient to extend m -way search trees by the addition of external nodes. An external (or failure) node is added wherever we otherwise have a NULL pointer. An external node represents a node that can be reached during a search only if the element being sought is not in the tree. Nodes that are not external nodes are called internal nodes.

Definition: A *B-tree of order m* is an m -way search tree that either is empty or satisfies the following properties:

610 Multitway Search Trees

- (1) The root node has at least two children.
- (2) All nodes other than the root node and external nodes have at least $\lceil m/2 \rceil$ children.
- (3) All external nodes are at the same level. \square

Observe that when $m = 3$, all internal nodes of a B-tree have a degree that is either 2 or 3 and when $m = 4$, the permissible degrees for these nodes are 2, 3 and 4. For this reason, a B-tree of order 3 is known as a 2-3 tree and a B-tree of order 4 is known as a 2-3-4 tree. A B-tree of order 5 is not a 2-3-4-5 tree as a B-tree of order 5 cannot have nodes whose degree is 2 (except for the root). Also, notice that all B-trees of order 2 are full binary trees. Hence, B-trees of order 2 exist only when the number of key values is $2^k - 1$ for some k . However, for any $n \geq 0$ and any $m \geq 2$, there is always a B-tree of order m that contains n keys.

Figure 11.2 shows a 2-3 tree (i.e., a B-tree of order 3) and Figure 11.3 shows a 2-3-4 tree (i.e., a B-tree of order 4). Notice that each (internal) node of a 2-3 tree can hold 2 elements while each such node of a 2-3-4 tree can hold 3 elements. In the figures, only the keys are shown. Note also that although Figures 11.2 and 11.3 show external nodes, external nodes are introduced only to make it easier to define and talk about B-trees. External nodes are not physically represented inside a computer. Rather, the corresponding child pointer of the parent of each external node is set to NULL.



Figure 11.2: Example of a 2-3 tree



Figure 11.3: Example of a 2-3-4 tree

11.2.2 Number of Elements in a B-Tree

A B-tree of order m in which all external nodes are at level $l+1$ has at most $m^l - 1$ keys. What is the minimum number, N , of elements in such a B-tree? From the definition of a B-tree we know that if $l > 1$, the root node has at least two children. Hence, there are at least two nodes at level 2. Each of these nodes must have at least $\lceil m/2 \rceil$ children. Thus, there are at least $2\lceil m/2 \rceil$ nodes at level 3. At level 4 there must be at least $2\lceil m/2 \rceil^2$ nodes, and continuing this argument, we see that there are at least $2\lceil m/2 \rceil^{l-1}$ nodes at level l when $l > 1$. All of these nodes are internal nodes. If the keys in the tree are K_1, K_2, \dots, K_N and $K_i < K_{i+1}$, $1 \leq i < N$, then the number of external nodes is $N + 1$. This is so because failures occur for $K_i < x < K_{i+1}$, $0 \leq i \leq N$, where $K_0 = -\infty$ and $K_{N+1} = +\infty$. This results in $N + 1$ different nodes that one could reach while searching for a key x that is not in the B-tree. Therefore, we have

$$\begin{aligned} N + 1 &= \text{number of external nodes} \\ &= \text{number of nodes at level } (l + 1) \\ &\geq 2\lceil m/2 \rceil^{l-1} \end{aligned}$$

so $N \geq 2\lceil m/2 \rceil^{l-1} - 1$, $l \geq 1$.

This in turn implies that if there are N elements (equivalently, keys) in a B-tree of order m , then all internal nodes are at levels less than or equal to l , $l \leq \log_{\lceil m/2 \rceil} [(N + 1)/2] + 1$. If a B-tree node can be examined with a single

612 Multiway Search Trees

memory access, the maximum number of accesses that have to be made for a search is l . Using a B-tree of order $m = 200$, which is quite practical for a disk resident B-tree, a tree with $N \leq 2 \times 10^6 - 2$ will have $l \leq \log_{200}((N + 1)/2) + 1$. Since l is an integer, we obtain $l \leq 3$. For $N \leq 2 \times 10^6 - 2$ we get $l \leq 4$.

To search a B-tree with a number of memory accesses equal to the B-tree height we must be able to examine a B-tree node with a single memory access. This means that the size of a node should not exceed the size of a memory block (i.e., size of a cache line or disk block). For main-memory resident B-trees an m in the tens is practical and for disk resident B-trees an m in the hundreds is practical.

11.2.3 Insertion into a B-Tree

The insertion algorithm for B-trees first performs a search to determine the leaf node, p , into which the new key is to be inserted. If the insertion of the new key into p results in p having m keys, the node p is split. Otherwise, the new p is written to the disk, and the insertion is complete. To split the node, assume that following the insertion of the new element, p has the format

$$m, A_0, (E_1, A_1), \dots, (E_m, A_m), \text{ and } E_i < E_{i+1}, 1 \leq i < m$$

The node is split into two nodes, p and q , with the following formats:

$$\text{node } p: \lceil m/2 \rceil - 1, A_0, (E_1, A_1), \dots, (E_{\lceil m/2 \rceil - 1}, A_{\lceil m/2 \rceil - 1}) \quad (11.5)$$

$$\text{node } q: m - \lceil m/2 \rceil, A_{\lceil m/2 \rceil}, (E_{\lceil m/2 \rceil + 1}, A_{\lceil m/2 \rceil + 1}), \dots, (E_m, A_m)$$

The remaining element, $E_{\lceil m/2 \rceil}$, and a pointer to the new node, q , form a tuple $(E_{\lceil m/2 \rceil}, q)$. This is to be inserted into the parent of p .

Inserting into the parent may require us to split the parent, and this splitting process can propagate all the way up to the root. When the root splits, a new root with a single element is created, and the height of the B-tree increases by one. A high-level description of the insertion algorithm for a disk resident B-tree is given in Program 11.2.

Example 11.1: Consider inserting an element with key 70 into the 2-3 tree of Figure 11.2. First we search for this key. If the key is already in the tree, then the existing element with this key is replaced by the new element. Since 70 is not in our example 2-3 tree, the new element is inserted and the total number of elements in the tree increases by 1. For the insertion, we need to know the leaf node encountered during the search for 70. Note that whenever we search for a key that is not in the 2-3 tree, the search encounters a unique leaf node. The leaf node encountered during the search for 70 is the node C, with key 80. Since this

```

// Insert element  $x$  into a disk resident B-tree.
Search the B-tree for an element  $E$  with key  $x.K$ .
If such an  $E$  is found, replace  $E$  with  $x$  and return;
Otherwise, let  $p$  be the leaf into which  $x$  is to be inserted;
 $q = \text{NULL}$ ;
for ( $e = x$ ;  $p$ ;  $p = p \rightarrow \text{parent}()$ )
{ // ( $e, q$ ) is to be inserted into  $p$ 
  Insert ( $e, q$ ) into appropriate position in node  $p$ ;
  Let the resulting node have the form:  $a, A_0, (E_1, A_1), \dots, (E_n, A_n)$ ;
  If ( $n \leq m - 1$ ) { // resulting node is not too big
    write node  $p$  to disk; return;
  }
  // node  $p$  has to be split
  Let  $p$  and  $q$  be defined as in Eq. (11.5);
   $e = E_{\lfloor m/2 \rfloor}$ ;
  write nodes  $p$  and  $q$  to the disk;
}
// a new root is to be created
Create a new node  $r$  with format 1, root, ( $e, q$ );
 $\text{root} = r$ ;
write root to disk;

```

Program 11.2: Insertion into a B-tree

node has only one element, the new element may be inserted here. The resulting 2-3 tree is shown in Figure 11.4(a).

Next, consider inserting an element with key 30. This time the search encounters the leaf node B. Since B is full, it is necessary to split B. For this, we first symbolically insert the new element into B to get the key sequence 10, 20, 30. Then the overfull node is split using Eq. 11.5. Following the split, B has the key sequence 10 and the new node, D, has 30. The middle element, whose key is 20, together with a pointer to the new node D is inserted into the parent A of B. The resulting 2-3 tree is shown in Figure 11.4(b).

Finally, consider the insertion of an element with key 60 into the 2-3 tree of Figure 11.4(b). The leaf node encountered during the search for 60 is node C. Since C is full, a new node, E, is created. Node E contains the element with the largest key (80). Node C contains the element with the smallest key (60). The element with the median key (70), together with a pointer to the new node, E, is inserted into the parent A of C. Again, since A is full, a new node, F, containing the element with the largest key among (30, 40, 70) is created. As

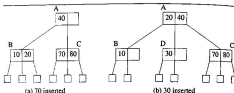


Figure 11.4: Insertion into the 2-3 tree of Figure 11.2

before, A contains the element with the smallest key. B and D remain the left and middle children of A, respectively, and C and E become these children of F. If A had a parent, then the element with the median key 40 and a pointer to the new node, F, would be inserted into this parent node. Since A does not have a parent, we create a new root, G, for the 2-3 tree. Node G contains the element with key 40, together with child pointers to A and F. The new 2-3 tree is as shown in Figure 11.3. \square

Analysis of B-tree Insertion: For convenience, assume the B-tree is disk resident. If h is the height of the B-tree, then h disk accesses are made during the top-down search. In the worst case, all h of the accessed nodes may split during the bottom-up splitting pass. When a node other than the root splits, we need to write out two nodes. When the root splits, three nodes are written out. If we assume that the h nodes read in during the top-down pass can be saved in memory so that they are not to be retrieved from disk during the bottom-up pass, then the number of disk accesses for an insertion is at most h (downward pass) + $2(h - 1)$ (nonroot splits) + 3 (root split) = $3h + 1$.

The average number of disk accesses is, however, approximately $h + 1$ for large m . To see this, suppose we start with an empty B-tree and insert N values into it. The total number of nodes split is at most $p - 2$, where p is the number of internal nodes in the final B-tree with N entries. This upper bound of $p - 2$ follows from the observation that each time a node splits, at least one additional node is created. When the root splits, two additional nodes are created. The first node created results from no splitting, and if a B-tree has more than one node, then the root must have split at least once. Figure 11.6 shows that $p - 2$ is the tightest upper bound on the number of nodes split in the creation of a p -node B-

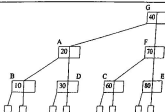


Figure 11.5: Insertion of 60 into the 2-3 tree of Figure 11.4(b)

tree when $p > 2$ (note that there is no B-tree with $p = 2$). A B-tree of order m with p nodes has at least $1 + (\lfloor m/2 \rfloor - 1)(p - 1)$ keys, as the root has at least one key and remaining nodes have at least $\lfloor m/2 \rfloor - 1$ keys each. The average number of splits, s_{avg} , may now be determined as follows:

$$\begin{aligned} s_{\text{avg}} &= (\text{total number of splits})/N \\ &\leq (p - 2)(1 + (\lfloor m/2 \rfloor - 1)(p - 1)) \\ &< 1(\lfloor m/2 \rfloor - 1) \end{aligned}$$

For $m = 200$ this means that the average number of node splits is less than 1/99 per key inserted. The number of disk accesses in an insertion is $k + 2s - 1$, where s is the number of nodes that are split during the insertion. So, the average number of disk accesses is $k + 2s_{\text{avg}} + 1 < k + 101/99 = k + 1$. \square

11.2.4 Deletion from a B-Tree

For convenience, assume we are deleting from a B-tree that resides on disk. Suppose we are to delete the element whose key is x . First, we search for this key. If x is not found, no element is to be deleted. If x is found in a node, z , that is not a leaf, then the position occupied by the corresponding element in z is filled by an element from a leaf node of the B-tree. Suppose that x is the i th key

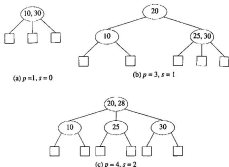


Figure 11.6: B-trees of order 3

in z (i.e., $s = E_j K$). Then E_j may be replaced by either the element with smallest key in the subtree A_j or the element with largest key in the subtree A_{j+1} . Both of these elements are in leaf nodes. In this way the deletion from a nonleaf node is transformed into a deletion from a leaf. For example, if we are to delete the element with key 20 that is in the root of Figure 11.6 (c), then this element may be replaced by either the element with key 10 or the element with key 25. Both are in leaf nodes. Once the replacement is done, we are faced with the problem of deleting either the 10 or the 25 from a leaf.

There are four possible cases when deleting from a leaf node p . In the first, p is also the root. If the root is left with at least one element, the changed root is written to disk and we are done. Otherwise, the B-tree is empty following the deletion. In the remaining cases, p is not the root. In the second case, following the deletion, p has at least $\lceil m/2 \rceil - 1$ elements. The modified leaf is written to disk, and we are done.

In the third case (rotation), p has $\lceil m/2 \rceil - 2$ elements, and its nearest

sibling, q , has at least $\lceil m/2 \rceil$ elements. To determine this, we examine only one of the two (at most) nearest siblings that p may have. p is deficient, as it has one less than the minimum number of elements required. q has more elements than the minimum required. A rotation is performed. In this rotation, the number of elements in q decreases by one, and the number in p increases by one. As a result, neither p nor q is deficient following the rotation. The rotation leaves behind a valid B-tree. Let r be the parent of p and q . If q is the nearest right sibling of p , then let i be such that E_i is the i th element in r ; all elements in p have a key that is less than E_i, K , and all those in q have a key that is greater than E_i, K . For the rotation, E_i becomes the rightmost element in p , E_i is replaced, in r , by the first (i.e., smallest) element in q , and the leftmost subtree of q becomes the rightmost subtree of p . The changed nodes p , q , and r are written to disk, and the deletion is complete. The case when q is the nearest left sibling of p is similar.

Figure 11.7 shows the rotation cases for a 3-3 tree. A “?” denotes a situation in which the presence or absence of an element is irrelevant. a , b , c , and d denote the children (i.e., roots of subtrees) of nodes.

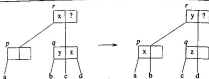
In the fourth case (combine) for deletion, p has $\lceil m/2 \rceil - 2$ elements, and its nearest sibling q has $\lceil m/2 \rceil - 1$ elements. So, p is deficient, and q has the minimum number of elements required by a nonroot node. Now, nodes p and q and the in-between element E_i in the parent r are combined to form a single node. The combined node has $(\lceil m/2 \rceil - 2) + (\lceil m/2 \rceil - 1) + 1 = 2\lceil m/2 \rceil - 2 \leq m - 1$ elements, which will, at most, fill the node. The combined node is written to disk. The combining operation reduces the number of elements in the parent node, r , by one. If the parent does not become deficient (i.e., it has at least one element if it is the root and at least $\lceil m/2 \rceil - 1$ elements if it is not the root), the changed parent is written to disk, and we are done. Otherwise, if the deficient parent is the root, it is discarded, as it has no elements. If the deficient parent is not the root, it has exactly $\lceil m/2 \rceil - 2$ elements. To remove this deficiency, we first attempt a rotation with one of r 's nearest siblings. If this is not possible, a combine is done. This process of combining can continue up the B-tree only until the children of the root are combined.

Figure 11.8 shows the two cases for a combine in a 3-3 tree when p is the left child of r . We leave it as an exercise to obtain the figures for the cases when p is a middle child and when p is a right child.

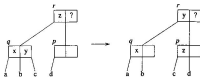
A high-level description of the deletion algorithm is provided in Program 11.3.

Example 11.2: Let us begin with the 3-3 tree of Figure 11.9(a). Suppose that the two element fields in a node of a 3-3 tree are called *data1* and *data2*. To delete the element with key 70, we must merely delete this element from node C. The result is shown in Figure 11.9(b). To delete the element with key 10 from

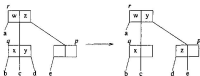
618 Multiway Search Trees



(a) p is the left child of r



(b) p is the middle child of r



(c) p is the right child of r

Figure 11.7: The three cases for rotation in a 2-3 tree

the 2-3 tree of Figure 11.9(b), we need to shift $dataR$ to $dataL$ in node B. This results in the 2-3 tree of Figure 11.9(c).

Next consider the deletion of the element with key 60. This leaves node C deficient. Since the right sibling, D, of C has 3 elements, we are in case 3 and a

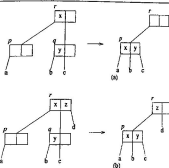


Figure 11.8: Combining in a 2-3 tree when p is the left child of r

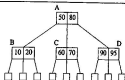
rotation is performed. In this rotation, we move the in-between element (i.e., the element whose key is 80) of the parent A of C and D to the *data* position of C and move the smallest element of D (i.e., the element whose key is 20) into the in-between position of the parent A of C and D (i.e., the *data* position of A). The resulting 2-3 tree takes the form shown in Figure 11.9(d). When the element with key 95 is deleted, node D becomes deficient. The rotation performed when the 60 was deleted is not possible now, as the left sibling, C , has the minimum number of elements required by a node in a B-tree of order 3. We now are in case 4 and must combine nodes C and D and the in-between element (90) in the parent A of C and D . For this, we move the 90 into the left sibling, C , and delete node D . Notice that in a combine, one node is deleted, whereas in a rotation, no node is deleted. The deletion of 95 results in the 2-3 tree of Figure 11.9(e). Deleting the element with key 90 from this tree results in the 2-3 tree of Figure 11.9(f). Now consider deleting the element with key 20 from this tree. Node B becomes deficient. At this time, we examine B 's right sibling, C . If C has excess

```

// Delete element with key  $x$ .
Search the B-tree for the node  $p$  that contains the element whose key is  $x$ ;
if there is no such  $p$  return; // no element to delete
Let  $p$  be of the form  $n, A_0, (E_1, A_1), \dots, (E_m, A_m)$  and  $E_i.k = x$ ;
if  $p$  is not a leaf {
    Replace  $E_i$  with the element with the smallest key in subtree  $A_i$ ;
    Let  $p$  be the leaf of  $A_i$  from which this smallest element was taken;
    Let  $p$  be of the form  $n, A_0, (E_1, A_1), \dots, (E_s, A_s)$ ;
     $i = 1$ ;
}
// delete  $E_i$  from node  $p$ , a leaf
Delete  $(E_i, A_i)$  from  $p$ ;  $s = s - 1$ ;
while  $((s < \lceil m/2 \rceil - 1) \text{ \&\& } p \neq \text{root})$ 
    if  $p$  has a nearest right sibling  $q$  {
        Let  $q = n_q, A'_0, (E'_1, A'_1), \dots, (E'_{s'}, A'_{s'})$ ;
        Let  $r = n_r, A'_0, (E'_1, A'_1), \dots, (E'_{s'}, A'_{s'})$  be the parent of  $p$  and  $q$ ;
        Let  $A'_i = q$  and  $A'_{i-1} = p$ ;
        if  $(n_q > \lceil m/2 \rceil)$  //  $\delta$  rotation
             $(E_{s+1}, A_{s+1}) = (E'_s, A'_s)$ ;  $s = s + 1$ ; // update node  $p$ 
             $E'_i = E'_i$ ; // update node  $r$ 
             $(n_q, A'_0, (E'_1, A'_1), \dots) = (n_r - 1, A'_1, (E'_2, A'_2), \dots)$ ;
            // update node  $q$ 
            write nodes  $p, q$  and  $r$  to disk; return;
    } // end of rotation
    // combine  $p, E'_i$ , and  $q$ 
     $s = 2 * \lceil m/2 \rceil - 2$ ;
    write  $n, A_0, (E_1, A_1), \dots, (E_m, A_m), (E'_i, A'_i), (E'_1, A'_1), \dots, (E'_{s'}, A'_{s'})$ 
    to disk as node  $p$ ;
    // update for next iteration
     $(n, A_0, \dots) = (n_r - 1, A'_0, \dots, (E'_{i-1}, A'_{i-1}), (E'_{i+1}, A'_{i+1}), \dots)$ 
     $p = r$ ;
} // end of if  $p$  has a nearest right sibling
else // node  $p$  must have a left sibling
    // this is symmetric to the case where  $p$  has a right sibling.
    // and is left as an exercise
} // end of if-else and while
if  $(n)$  write  $p, (n, A_0, \dots, (E_s, A_s))$ 
else root =  $A_0$ ; // new root

```

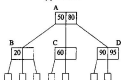
Program 11.3: Deletion from a B-tree that resides on disk



(a) Initial 2-3 tree



(b) 70 deleted



(c) 10 deleted

Figure 11.9: Deletion from a 2-3 tree (continued on next page)

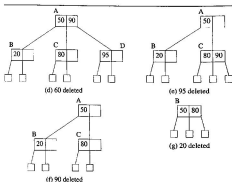


Figure 11.9: Deletion from a 3-3 tree

elements, we can perform a rotation similar to that done during the deletion of 60. Otherwise, a combine is performed. Since C doesn't have excess elements, we proceed in a manner similar to the deletion of 95 and do a combine. This time the elements with keys 50 and 80 are moved into B, and node C is deleted. This, however, causes the parent node A to become deficient. If the parent had not been a root, we would examine its left or right sibling, as we did when nodes C (deletion of 60) and D (deletion of 95) became empty. Since A is the root, it is simply deleted, and B becomes the new root (Figure 11.9(g)). Recall that a root is deficient iff it has no element. □

Analysis of B-tree Deletion: Once again, we assume a disk-resident B-tree and that disk nodes accessed during the downward search pass may be saved in a stack in main memory, so they do not need to be reaccessed from the disk during

the upward restructuring pass. For a B-tree of height h , h disk accesses are made to find the node from which the key is to be deleted and to transform the deletion to a deletion from a leaf. In the worst case, a combine takes place at each of the last $h - 2$ nodes on the root-to-leaf path, and a rotation takes place at the second node on this path. The $h - 2$ combines require this many disk accesses to retrieve a nearest sibling for each node and another $h - 2$ accesses to write out the combined nodes. The rotation requires one access to read a nearest sibling and three to write out the three nodes that are changed. The total number of disk accesses is $3h$.

The deletion time can be reduced at the expense of disk space and a slight increase in node size by including a delete bit, F_i , for each element, E_i , in a node. Then we can set $F_i = 1$ if E_i has not been deleted and $F_i = 0$ if it has. No physical deletion takes place. In this case a delete requires a maximum of $h + 1$ accesses (h to locate the node containing the element to be deleted and 1 to write out this node with the appropriate delete bit set to 0). With this strategy, the number of nodes in the tree never decreases. However, the space used by deleted entries can be reused during further insertions (see Exercises). As a result, this strategy has little effect on search and insert times (the number of levels increases very slowly when n is large). The time taken to insert an item may even decrease slightly because of the ability to reuse deleted element space. Such reuses would not require us to split any nodes. \square

EXERCISES

1. Show that all B-trees of order 2 are full binary trees.
2. Use the insertion algorithm of Program 11.2 to insert an element with key 40 into the 2-3 tree of Figure 11.9(a). Show the resulting 2-3 tree.
3. Use the insertion algorithm of Program 11.2 to insert elements with keys 43, 95, 96, and 97, in this order, into the 2-3-4 tree of Figure 11.3. Show the resulting 2-3-4 tree following each insert.
4. Use the deletion algorithm of Program 11.3 to delete the elements with keys 90, 95, 80, 70, 60 and 50, in this order, from the 2-3 tree of Figure 11.9(a). Show the resulting 2-3 tree following each deletion.
5. Use the deletion algorithm of Program 11.3 to delete the elements with keys 85, 90, 92, 75, 60, and 70 from the 2-3-4 tree of Figure 11.3. Show the resulting 2-3-4 tree following each deletion.
6. (a) Insert elements with keys 62, 5, 83, and 75 one at a time into the order-3 B-tree of Figure 11.10. Show the new tree after each element is inserted. Do the insertion using the insertion process described in the text.
(b) Assuming that the tree is kept on a disk and one node may be

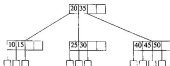


Figure 11.10: B-tree of order 5

- retrieved at a time, how many disk accesses are needed to make each insertion? State any assumptions you make.
- (c) Delete the elements with keys 45, 40, 10, and 25 from the order-5 B-tree of Figure 11.10. Show the tree following each deletion. The deletions are to be performed using the deletion process described in the text.
- (d) How many disk accesses are made for each of the deletions?
- Complete Figure 11.8 by adding figures for the cases when p is the middle child and right child of its parent.
 - Complete the symmetric case of Program 11.3.
 - Develop the C++ class `TwoThreeTree`, which implements a 2-3 tree. You must include functions to search, insert and delete. Test your class using your own test data.
 - Develop the C++ class `TwoThreeFourTree`, which implements a 2-3-4 tree. You must include functions to search, insert and delete. Test your class using your own test data.
 - Write insertion and deletion algorithms for B-trees assuming that with each element is associated an additional data member, *deleted*, such that *deleted* = *false* iff the corresponding element has not been deleted. Deletions should be accomplished by setting *deleted* = *false*, and insertions should make use of deleted space whenever possible without restructuring the tree.

12. Write algorithms to search and delete keys from a B-tree by position; that is, $\text{Get}(k)$ finds the k th smallest key, and $\text{Delete}(k)$ deletes the k th smallest key in the tree. (Hint: To do this efficiently, additional information must be kept in each node. With each pair (E_i, A_i) keep $N_i = \sum_{j=1}^i (n_j + 1)$.) What are the worst-case computing times of your algorithms?
 13. The text assumed a node structure that was sequential. However, we need to perform the following functions on a B-tree node: search, insert, delete, join, and split.
 - (a) Explain why each of these functions is important during a search, insert, and delete operation in the B-tree.
 - (b) Explain how a red-black tree could be used to represent each node. You will need to use integer pointers and regard each red-black tree as embedded in an array.
 - (c) What kind of performance gain/loss do you expect using red-black trees for each node instead of a sequential organization? Try to quantify your answer.
 14. Modify Program 11.2 so that when node p has m elements, we first check to see if either the nearest left sibling or the nearest right sibling of p has fewer than $m - 1$ elements. If so, p is not split. Instead, a rotation is performed moving either the smallest or largest element in p to its parent. The corresponding element in the parent, together with a subtree, is moved to the sibling of p that has space for another element.
 15. [Bayer and McCreight] Suppose that an insertion has been made into node p and that it has become over-full (i.e., it now contains m elements). Further, suppose that its nearest right sibling q is full (i.e., it contains $m - 1$ elements). So, the elements in p and q together with the in-between element in the parent of p and q make $2m$ elements. These $2m$ elements may be partitioned into three nodes p , q , and r containing $\lfloor (2m - 2)/3 \rfloor$, $\lfloor (2m - 1)/3 \rfloor$, and $\lfloor 2m/3 \rfloor$ elements, respectively, plus two in-between elements (one for p and q and the other for q and r). So, we may split p and q into 3 nodes p , q , and r that are almost two-thirds full, replace the former in-between element for p and q with the new one, and then insert the in-between element for q and r together with a pointer to the new node r into the parent of p and q . The case when q is the nearest left sibling of p is similar.
- Rewrite Program 11.2 so that node splittings occur only as described here.
16. A B*-tree of order m is a search tree that either is empty or satisfies the following properties:
 - (a) The root node has at least two and at most $2\lfloor (2m - 2)/3 \rfloor + 1$

children.

- (b) The remaining internal nodes have at most m and at least $\lceil(2m - 1)/3\rceil$ children each.

- (c) All external nodes are on the same level.

For a B^+ -tree of order m that contains N elements, show that if $x = \lceil(2m - 1)/3\rceil$, then

- (a) the height, h , of the B^+ -tree satisfies $h \leq 1 + \log_x \lceil(N + 1)/2\rceil$
 (b) the number of nodes p in the B^+ -tree satisfies $p \leq 1 + (N - 1)/(x - 1)$

What is the average number of splits per insert if a B^+ -tree is built up starting from an empty B^+ -tree?

17. Using the splitting technique of Exercise 15, write an algorithm to insert a new element, x , into a B^+ -tree of order m . How many disk accesses are made in the worst case and on the average? Assume that the B -tree was initially of depth 1 and that it is maintained on a disk. Each access retrieves or writes one node.
18. Write an algorithm to delete the element whose key is x from a B^+ -tree of order m . What is the maximum number of accesses needed to delete from a B^+ -tree of depth l ? Make the same assumptions as in Exercise 17.

11.3 B^+ -TREES

11.3.1 Definition

A B^+ -tree is a close cousin of the B -tree. The essential differences are:

- (1) In a B^+ -tree we have two types of nodes—index and data. The index nodes of a B^+ -tree correspond to the internal nodes of a B -tree while the data nodes correspond to external nodes. The index nodes store keys (not elements) and pointers and the data nodes store elements (together with their keys but no pointers).
- (2) The data nodes are linked together, in left to right order, to form a doubly linked list.

Figure 11.11 gives an example B^+ -tree of order 3. The data nodes are shaded while the index nodes are not. Notice that the index nodes form a 2-3 tree whose height is 2. The capacity of a data node need not be the same as that of an index node. In Figure 11.11 each data node can hold 3 elements while each index

node can hold 2 keys.

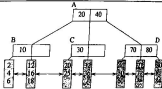


Figure 11.11: A B⁺-tree of order 3

Definition: A B⁺-tree of order m is a tree that either is empty or satisfies the following properties:

- (1) All data nodes are at the same level and are leaves. Data nodes contain elements only.
- (2) The index nodes define a B-tree of order m ; each index node has keys but not elements.
- (3) Let

$$s, A_0, (K_1, A_1), (K_2, A_2), \dots, (K_n, A_n)$$

where the A_i , $0 \leq i \leq n$, are pointers to subtrees, and the K_i , $1 \leq i \leq n$, are keys in the format of some index node. Let $K_0 = -\infty$ and $K_{n+1} = \infty$. All elements in the subtree A_i have key less than K_{i+1} and greater than or equal to K_i , $0 \leq i \leq n$. \square

11.3.2 Searching a B⁺-Tree

B⁺-trees support two types of searches—exact match and range. To search the tree of Figure 11.11 for the element whose key is 32, we begin at the root A , which is an index node. From the definition of a B⁺-tree we know that all elements in the left subtree of A (i.e., the subtree whose root is B) have a key

628 Multiway Search Trees

smaller than 20; those in the subtree with root C have keys ≥ 20 and < 40 ; and those in the subtree with root D have keys ≥ 40 . So, the search moves to the index node C. Since the search key is ≥ 30 , the search moves from C to the data node that contains the elements with keys 32 and 36. This data node is searched and the desired element reported. Program 11.4 gives a high-level description of the algorithm to search a B^+ -tree.

```
// Search a  $B^+$ -tree for an element with key  $x$ .
// Return the element if found. Return NULL otherwise.
// If the tree is empty return NULL;
 $K_0 = -\text{MAXKEY}$ ;
for ( $*p = \text{root}$ ;  $p$  is an index node;  $p = A_i$ )
{
    Let  $p$  have the format  $s, A_0, (K_1, A_1), \dots, (K_n, A_n)$ ;
     $K_{n+1} = \text{MAXKEY}$ ;
    Determine  $i$  such that  $K_i \leq x < K_{i+1}$ ;
}
// search the data node  $p$ 
Search  $p$  for an element  $E$  with key  $x$ ;
if such an element is found return  $E$ 
else return NULL;
```

Program 11.4: Searching a B^+ -tree

To search for all elements with keys in the range [16, 70], we proceed as in an exact match search for the start, 16, of the range. This gets us to the second data node in Figure 11.11. From here, we march down (rightward) the doubly linked list of data nodes until we reach a data node that has an element whose key exceeds the end, 70, of the search range (or until we reach the end of the list). In our example, 4 additional data nodes are examined. All examined data nodes other than the first and last contain at least one element that is in the search range.

11.3.3 Insertion into a B^+ -Tree

An important difference between inserting into a B-tree and inserting into a B^+ -tree is how we handle the splitting of a data node. When a data node becomes overfull, half the elements (those with the largest keys) are moved into a new node; the key of the smallest element so moved together with a pointer to the newly created data node are inserted into the parent index node (if any) using the

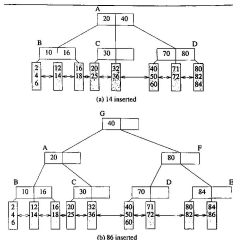
insertion procedure for a B-tree. The splitting of an index node is identical to the splitting of an internal node of a B-tree.

Consider inserting an element with key 27 into the B⁺-tree of Figure 11.11. We first search for this key. The search gets us to the data node that is the left child of C. Since this data node contains no element with key 27 and since this data node isn't full, we insert the new element as the third element in this data node. Next, consider the insertion of an element with key 14. The search for 14 gets us to the second data node, which is full. Symbolically inserting the new element into this full node results in an overfull node with the key sequence 12, 14, 16, 18. The overfull node is split into two by moving the larger half of the elements (those with keys 16 and 18) into a new data node, which is then inserted into the doubly linked list of data nodes. The smallest key, 16, in this new data node together with a pointer to the new data node are inserted in the parent index node B to get the configuration of Figure 11.12 (a).

Finally, consider inserting an element with key 86 into the B⁺-tree of Figure 11.12 (a). The search for 86 gets us to the rightmost data node, which is full. Symbolically inserting the new element into this node results in the key sequence 80, 82, 84, 86. Splitting the overfull data node creates a new data node with the elements whose keys are 84 and 86. The new data node is inserted into the doubly linked list of data nodes. Then we insert the key 84 and a pointer to the new data node into the parent index node D, which becomes overfull. The overfull D is split using Eq. 11.5. The 84 along with two of the 4 subtrees of the overfull D are moved into a new index node E and the 80 together with a pointer to E inserted into the parent A of D. This causes A to become overfull. The overfull A is split using Eq. 11.5 and we get a new index node F that has the key 80 and 2 of the 4 subtrees of the overfull A. The key 40 together with pointers to A and F form the new root of the B⁺-tree (Figure 11.12 (b)).

11.3.4 Deletion from a B⁺-Tree

Since elements are stored only in the leaves of a B⁺-tree, we need concern ourselves only with deletion from a leaf (recall that in the case of a B-tree we had to transform a deletion from a non-leaf into a deletion from a leaf; this case doesn't arise for B⁺-trees). Since the index nodes of a B⁺-tree form a B-tree, a non-root index node is deficient when it has fewer than $\lceil n/2 \rceil - 1$ keys and a root index node is deficient when it has no key. When is a data node deficient? The definition of a B⁺-tree doesn't specify a minimum occupancy for a data node. However, we may get some guidance from our algorithm to insert an element. Following the split of an overfull data node, the original data node as well as the new one each have at least $\lceil c/2 \rceil$ elements, where c is the capacity of a data node. So, except when a data node is the root of the B⁺-tree, its occupancy is at

Figure 11.12: Insertion into the B⁺-tree of Figure 11.11

least $\lceil c/2 \rceil$. We shall say that a non-root data node is deficient iff it has fewer than $\lceil c/2 \rceil$ elements; a root data node is deficient iff it is empty.

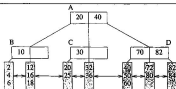
We illustrate the deletion process by an example. Consider the B⁺-tree of Figure 11.11. The capacity c of a data node is 3. So, a non-root data node is deficient iff it has fewer than 2 elements. To delete the element whose key is 40, we first search for the element to be deleted. This element is found in the data

node that is the left child of the index node D. Following the deletion of the element with key 40, the occupancy of this data node becomes 2. So, the data node isn't deficient and we need merely write the modified data node to disk (assuming the B⁺-tree is disk resident) and we are done. Notice that when the deletion of an element doesn't result in a deficient data node, no index node is changed.

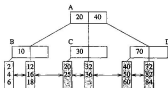
Next consider the deletion of the element whose key is 71 from the B⁺-tree of Figure 11.11. This element is found in the middle child of D. Following its deletion, the middle child of D becomes deficient. We check either its nearest right or nearest left sibling and determine whether the checked sibling has more than the required minimum number ($\lceil c/2 \rceil$) of elements. Suppose we check the nearest right sibling, which has the key sequence 80, 82, 84. Since this node has an excess element, we borrow the smallest and update the in-between key in the parent D from 80 to that of the smallest remaining element in the right sibling, 82. Figure 11.13 (a) shows the result. For a disk-resident B⁺-tree, this deletion would require us to write out one altered index node (D) and two altered data nodes. For data nodes with larger capacity, when a data node becomes deficient, we may borrow several elements from a nearest sibling that has excess elements. For example, when $c=10$, a deficient data node will have 4 elements and its nearest sibling may have 10. We could borrow 3 elements from the nearest sibling thereby balancing the occupancy in both data nodes to 7. Such a balancing is expected to improve performance.

When we delete the element with key 80 from the B⁺-tree of Figure 11.13 (a), the middle child of D becomes deficient. We check its nearest right sibling and discover that this sibling has only $\lceil c/2 \rceil$ elements. So, the 2 data nodes are combined into one and the in-between key 82 that is in the parent index node D deleted. Figure 11.13 (b) shows the resulting B⁺-tree. Notice that combining two data nodes into one requires the deletion of a data node from the doubly linked list of data nodes. Note also that in the case of a disk-resident B⁺-tree, the just performed deletion requires us to write out one altered data node (the middle child of D) and one altered index node (D).

As another example for deletion, consider deleting the element with key 32 from the B⁺-tree of Figure 11.12 (b). This element is in the middle child of C. Following the deletion, the middle child becomes deficient. Since its nearest sibling has only $\lceil c/2 \rceil$ elements, we cannot borrow from it. Instead, we combine the two data nodes deleting one from its doubly linked list and delete the in-between key (30) in the parent. Figure 11.14 (a) shows the result. As we can see, the index node C now has become deficient. When an index node becomes deficient, we examine a nearest sibling. If the examined nearest sibling has excess keys, we balance the occupancy of the two index nodes; this balancing involves moving some keys and associated subtrees as well as changing the in-between key in the parent. For our example, the in-between key 30 is moved from A to C, the rightmost key 16 of B is moved to A, and the right subtree of B



(a) 71 deleted from Figure 11.11

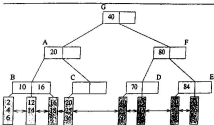


(b) 80 deleted from (a)

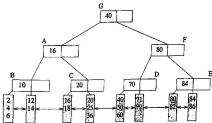
Figure 11.13: Deletion from a B^+ -tree

moved to C. Figure 11.14 (b) shows the resulting B^+ -tree.

As a final example, consider the deletion of the element with key 86 from the B^+ -tree of Figure 11.12 (b). The middle child of E becomes deficient and is combined with its sibling; a data node is deleted from the doubly linked list of data nodes and the in-between key 84 in the parent also is deleted. This results in a deficient index node E and the configuration of Figure 11.15 (a). The deficient index node E combines with its sibling index node D and the in-between key 80 to get the configuration of Figure 11.15 (b). Finally, the deficient index node F combines with its sibling A and the in-between key 40 in its parent G. This causes the parent G, which is the root, to become deficient. The deficient root is

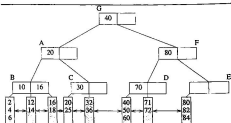


(a) C is deficient

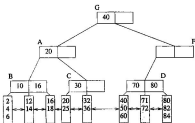


(b) After borrowing from B

Figure 11.14: Stages in deleting 32 from the B⁺ tree of Figure 11.12 (b)



(a) E becomes deficient



(b) F becomes deficient

Figure 11.15: Stages in deleting 86 from the B⁺-tree of Figure 11.12 (b)

discarded and we get the B⁺-tree of Figure 11.12 (a). In the case of a disk resident B⁺-tree, the deletion of 86 would require us to write to disk one altered data node and 2 altered index nodes (A and D).

EXERCISES

1. Into the B⁺-tree of Figure 11.11 insert elements with keys 5, 38, 45, 11 and 81 (in this order). Use the insertion method described in the text. Draw the B⁺-tree following each insert.
2. Provide a high-level description (similar to Program 11.4) of the algorithm to insert into a B⁺-tree.
3. Suppose that a B⁺-tree whose height is h is disk resident. How many disk accesses are needed, in the worst case, to insert a new element? Assume that each node may be read/written with a single access and that we have sufficient memory to save the h nodes accessed in the search phase so that these nodes don't have to be re-read during the bottom-up node splitting phase.
4. From the B⁺-tree of Figure 11.12 (b) delete the elements with keys 6, 71, 14, 18, 16 and 2 (in this order). Use the deletion method described in the text. Show the B⁺-tree following each delete.
5. Provide a high-level description (similar to Program 11.4) of the algorithm to delete from a B⁺-tree.
6. Suppose that a B⁺-tree whose height is h is disk resident. How many disk accesses are needed, in the worst case, to delete an element? Assume that each node may be read/written with a single access and that we have sufficient memory to save the h nodes accessed in the search phase so that these nodes don't have to be re-read during the bottom-up borrow and combine phase.
7. Discuss the merits/demerits of replacing the doubly linked list of data nodes in a B⁺-tree by a singly linked list.
8. Program the C++ class *BPlusTree* that implements a B⁺-tree. Your class must include functions for exact and range search as well as for insert and delete. Test all functions using your own test data.

11.4 REFERENCES AND SELECTED READINGS

B-trees were invented by Bayer and McCreight. For further reading on B-trees and their variants, see "Organization and maintenance of large ordered indices," by R. Bayer and E. McCreight, *Acta Informatica*, 1972; *The art of computer*

636 Multisearch Trees

programming, Vol. 3, *Sorting and Searching*, Second Edition, by D. Knuth, Addison Wesley, 1997; "The ubiquitous B-tree," by D. Comer, *ACM Computing Surveys*, 1979; and "B trees," by D. Zhang, in *Handbook of data structures and applications*, D. Mehta and S. Sahni editors, Chapman & Hall/CRC, 2005.

CHAPTER 12

Digital Search Structures

12.1 DIGITAL SEARCH TREES

12.1.1 Definition

A *digital search tree* is a binary tree in which each node contains one element. The element-to-node assignment is determined by the binary representation of the element keys. Suppose that we number the bits in the binary representation of a key from left to right beginning at one. Then bit one of 1000 is 1, and bits two, three, and four are 0. All keys in the left subtree of a node at level i have bit i equal to zero whereas those in the right subtree of nodes at this level have bit $i = 1$. Figure 12.1(a) shows a digital search tree. This tree contains the keys 1000, 0010, 1001, 0001, 1100, and 0000.

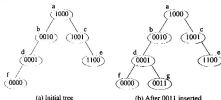


Figure 12.1: Digital search trees

12.1.2 Search, Insert and Delete

Suppose we are to search for the key $k = 0011$ in the tree of Figure 12.1(a). k is first compared with the key in the root. Since k is different from the key in the root, and since bit one of k is 0, we move to the left child, b , of the root. Now, since k is different from the key in node b , and bit two of k is 0, we move to the left child, d , of b . Since k is different from the key in node d and since bit three of k is one, we move to the right child of d . Node d has no right child to move to. From this we conclude that $k = 0011$ is not in the search tree. If we wish to insert k into the tree, then it is to be added as the right child of d . When this is done, we get the digital search tree of Figure 12.1(b).

The digital search tree functions to search and insert are quite similar to the corresponding functions for binary search trees. The essential difference is that the subtree to move to is determined by a bit in the search key rather than by the result of the comparison of the search key and the key in the current node. The deletion of an item in a leaf is done by removing the leaf node. To delete from any other node, the deleted item must be replaced by a value from any leaf in its subtree and that leaf removed.

Each of these operations can be performed in $O(k)$ time, where k is the height of the digital search tree. If each key in a digital search tree has *keySize* bits, then the height of the digital search tree is at most $\text{keySize} + 1$.

EXERCISES

1. Draw a different digital search tree than Figure 12.1 (a) that has the same set of keys.
2. Write the digital search tree functions for the search, insert, and delete operations. Assume that each key has *keySize* bits and that the function *bit(k, i)* returns the *i*th (from the left) bit of the key *k*. Show that each of your functions has complexity $O(h)$, where *h* is the height of the digital search tree.

12.2 BINARY TRIES AND PATRICIA

When we are dealing with very long keys, the cost of a key comparison is high. Since searching a digital search tree requires many comparisons between pairs of keys, digital search trees (and also binary and multiway search trees) are inefficient search structures when the keys are very long. We can reduce the number of key comparisons done during a search to one by using a related structure called *Patricia* (Practical algorithm to retrieve information coded in alphanumeric). We shall develop this structure in three steps. First, we introduce a structure called a *binary trie*. Then we transform binary tries into compressed binary tries. Finally, from compressed binary tries we obtain Patricia. Since binary tries and compressed binary tries are introduced only as a means of arriving at Patricia, we do not dwell much on how to manipulate these structures. A more general version of binary tries (called a *trie*) is considered in the next section.

12.2.1 Binary Tries

A *binary trie* is a binary tree that has two kinds of nodes: *branch nodes* and *element nodes*. A branch node has the two data members *leftChild* and *rightChild*. It has no data member. An element node has the single data member *data*. Branch nodes are used to build a binary tree search structure similar to that of a digital search tree. This search structure leads to element nodes.

Figure 12.2 shows a six-element binary trie. Element nodes are shaded. To search for an element with key *k*, we use a branching pattern determined by the bits of *k*. The *i*th bit of *k* is used at level *i*. If it is zero, the search moves to the left subtree. Otherwise, it moves to the right subtree. To search for 0010 we first follow the left child, then again the left child, and finally the right child.

Observe that a successful search in a binary trie always ends at an element node. Once this element node is reached, the key in this node is compared with

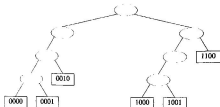


Figure 12.2: Example of a binary trie

the key we are searching for. This is the only key comparison that takes place. An unsuccessful search may terminate either at an element node or at a 0 pointer.

12.2.2 Compressed Binary Tries

The binary trie of Figure 12.2 contains branch nodes whose degree is one. By adding another data member, *bitNumber*, to each branch node, we can eliminate all degree-one branch nodes from the trie. The *bitNumber* data member of a branch node gives the bit number of the key that is to be used at this node. Figure 12.3 gives the binary trie that results from the elimination of degree-one branch nodes from the binary trie of Figure 12.2. The number outside a node is its *bitNumber*. A binary trie that has been modified in this way to contain no branch nodes of degree one is called a *compressed binary trie*.

12.2.3 Patricia

Compressed binary tries may be represented using nodes of a single type. The new nodes, called *augmented branch nodes*, are the original branch nodes augmented by the data member *data*. The resulting structure is called *Patricia* and

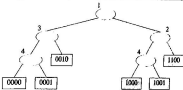


Figure 12.3: Binary trie of Figure 12.2 with degree-one codes eliminated

is obtained from a compressed binary trie in the following way:

- (1) Replace each branch node by an augmented branch node.
- (2) Eliminate the element nodes.
- (3) Store the data previously in the element nodes in the *data* data members of the augmented branch nodes. Since every nonempty compressed binary trie has one less branch node than it has element nodes, it is necessary to add one augmented branch node. This node is called the *header node*. The remaining structure is the left subtree of the header node. The header node has *bitNumber* equal to zero. Its *rightChild* data member is not used. The assignment of data to augmented branch nodes is done in such a way that the *bitNumber* in the augmented branch node is less than or equal to that in the parent of the element node that contained this data.
- (4) Replace the original pointers to element nodes by pointers to the respective augmented branch nodes.

When these transformations are performed on the compressed trie of Figure 12.3, we get the structure of Figure 12.4. Let *root* be the root of Patricia. *root* is 0 iff the Patricia is empty. A Patricia with one element is represented by a header node whose left-child data member points to itself (Figure 12.5(a)). We can distinguish between pointers that pointed originally to branch nodes and those that pointed to element nodes by noting that, in Patricia, the former pointers are directed to nodes with a greater *bitNumber* value, whereas pointers of the latter type are directed to nodes whose *bitNumber* value either is equal to or less than



Figure 12.4: An example of Patricia

that in the node where the pointer originates.

12.2.3.1 Searching Patricia

To search for an element with key k , we begin at the header node and follow a path determined by the bits in k . When an element pointer is followed, the key in the node reached is compared with k . This is the only key comparison made. No comparisons are made on the way down. Suppose we wish to search for $k = 0000$ in the Patricia instance of Figure 12.4. We begin at the header node and follow the left-child pointer to the node with 0000. The bit-number data member of this node is 1. Since bit one of k is 0, we follow the left child pointer to the node with 0010. Now bit three of k is used. Since this is 0, the search moves to the node with 0001. The bit-number data member of this node is 4. The fourth bit of k is zero, so we follow the left-child pointer. This brings us to a node with bit-number data member less than that of the node we moved from. Hence, an element pointer was used. Comparing the key in this node with k , we find a match, and the search is successful.

Next, suppose that we are to search for $k = 1011$. We begin at the header node. The search moves successively to the nodes with 0000, 1001, 1000, and 1001. k is compared with 1001. Since k is not equal to 1001, we conclude that there is no element with this key. The function to search a Patricia instance is given in Program 12.1. In this function, *PatNode* is the data type of the nodes in the tree and the function *bit(i, j)* returns the j th bit (the leftmost bit is bit one) of

i.

```

template <class K, class E>
E* Patricia<K, E>::Search(const K& k) const
{ // Search Patricia. Return a pointer to the element whose key is k.
  // Return NULL if no such element
  if (!root) return NULL; // Patricia is empty
  PatNode<K, E> *y = root->leftChild; // move to left child of header node
  for (PatNode<K, E> *p = root; y->bitNumber > p->bitNumber;)
    // follow a branch pointer
    p = y;
    if (bit(k, y->bitNumber)) y = y->rightChild;
    else y = y->leftChild;
  }
  // Check key in y
  if (y->key == k) return &y->element;
  return NULL;
}

```

Program 12.1: Searching Patricia

12.2.3.2 Inserting into Patricia

Let us now examine how we can insert new elements. Suppose we begin with an empty instance and wish to insert an element with key 1000. The result is an instance that has only a header node (Figure 12.5(a)). Next, consider inserting an element with key $k = 0010$. First, we search for this key using function *Patricia::Search* (Program 12.1). The search terminates at the header node. Since 0010 is not equal to the key $q = 1000$ in this node, we know that 0010 is not currently in the Patricia instance, so the element may be inserted. For this, the keys k and q are compared to determine the first (i.e., leftmost) bit at which they differ. This is bit one. A new node containing the element with key k is added as the left-child of the header node. Since bit one of k is zero, the left-child data member of this new node points to itself, and its right-child data member points to the header node. The bit-number data member is set to 1. The resulting Patricia instance is shown in Figure 12.5(b).

Suppose that the next element to be inserted has $k = 1001$. The search for this key ends at the node with $q = 1000$. The first bit at which k and q differ is bit $j = 4$. Now we search the instance of Figure 12.5(b) using only the first $j - 1 = 3$

bits of k . The last move is from the node with 0010 to that with 1000. Since this is a right-child move, a new node containing the element with key k is to be inserted as the right child of 0010. The bit-number data member of this node is set to $j = 4$. As bit four of k is 1, the right-child data member of the new node points to itself and its left-child data member points to the node with q . Figure 12.5(c) shows the resulting structure.

To insert $k = 1100$ into Figure 12.5(c), we first search for this key. Once again, $q = 1000$. The first bit at which k and q differ is $j = 2$. The search using only the first $j - 1$ bits ends at the node with 1001. The last move is a right child move from 0010. A new node containing the element with key k and bit-number data member $j = 2$ is added as the right child of 0010. Since bit j of k is one, the right-child data member of the new node points to itself. Its left-child data member points to the node with 1001 (this was previously the right child of 0010). The new Patricia instance is shown in Figure 12.5(d). Figure 12.5(e) shows the result of inserting an element with key 0000, and Figure 12.5(f) shows the Patricia instance following the insertion of 0001.

The preceding discussion leads to the insertion function *Patricia::insert* (Program 12.2). Its complexity is $O(h)$, where h is the height of the Patricia. h can be as large as $\min[\text{keySize} + 1, n]$, where *keySize* is the number of bits in a key and n is the number of elements. When the keys are uniformly distributed, the height is $O(\log n)$. We leave the development of the deletion function as an exercise.

EXERCISES

1. Write the binary trie functions for the search, insert, and delete operations. Assume that each key has *keySize* bits and that the function *bit*(k, i) returns the i th (from the left) bit of the key k . Show that each of your functions has complexity $O(h)$, where h is the height of the binary trie.
2. Write the compressed binary trie functions for the search, insert, and delete operations. Assume that each key has *keySize* bits and that the function *bit*(k, i) returns the i th (from the left) bit of the key k . Show that each of your functions has complexity $O(h)$, where h is the height of the compressed binary trie.
3. Write a function to delete the element with key k from a Patricia. The complexity of your function should be $O(h)$, where h is the height of the Patricia instance. Show that this is the case.

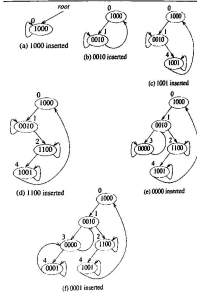


Figure 12.5: Insertion into Patricia

```

template <class K, class E>
void Patricia<K, E>::Insert(const K& k, const E& e)
// Insert e into the Patricia tree. k is the key.
    if (!root) { // Patricia is empty
        root = new PatNode<K, E>(0, k, e);
        // Create a PatNode and set its bitNumber, key and element fields
        root->leftChild = root; return;
    }
    PatNode<K, E> *y = NSearch(k);
    // NSearch returns pointer to last node seen in search for k
    if (y->key == k) { y->element = e; return; }
    // Update old element

    // New element. A new node with e is to be inserted
    // Find first bit where k and y->key differ
    for (int j = 1; bit(k, j) == bit(y->key, j); j++) ;

    // Search Patricia using first j - 1 bits of k
    PatNode<K, E> *x = root->leftChild, *p = root;
    while ((x->bitNumber > p->bitNumber) && (x->bitNumber < j)) {
        p = x;
        if (bit(k, x->bitNumber)) x = x->leftChild;
        else x = x->rightChild;
    }

    // Insert x as a child of p
    PatNode<K, E> *z = new PatNode<K, E>(j, k, e);
    if (bit(k, j)) { z->leftChild = x; z->rightChild = x; }
    else { z->leftChild = x; z->rightChild = x; }

    if (x == p->leftChild) p->leftChild = z;
    else p->rightChild = z;
    return;
}

```

Program 12.2: Insertion into Patricia

12.3 MULTIWAY TRIES

12.3.1 Definition

A multiway trie (or, simply, trie) is a structure that is particularly useful when key values are of varying size. This data structure is a generalization of the binary trie that was developed in the preceding section.

A *trie* is a tree of degree $m \geq 2$ in which the branching at any level is determined not by the entire key value, but by only a portion of it. As an example, consider the trie of Figure 12.6 in which the keys are composed of lowercase letters from the English alphabet. The trie contains two types of nodes: *element*, and *branch*. In Figure 12.6, element nodes are shaded while branch nodes are not shaded. An element node has only a *data* field; a branch node contains pointers to subtrees. In Figure 12.6, each branch node has 27 pointer fields. The extra pointer field is used for the blank character (denoted B). This character is used to terminate all keys, as a trie requires that no key be a prefix of another (see Figure 12.7).

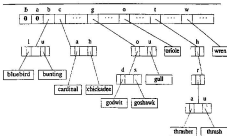


Figure 12.6: Trie created using characters of key value from left to right, one at a time.

At the first level all key values are partitioned into disjoint classes depending on their first character. Thus, $root \rightarrow child[i]$ points to a subtree containing all key values beginning with the i th letter. On the j th level the branching is



Figure 12.7: Trie showing need for a terminal character (in this case a blank)

determined by the *jth* character. When a subtree contains only one key value, it is replaced by a node of type element. This node contains the key value, together with other relevant information, such as the address of the record with this key value.

As another example of a trie, suppose that we have a collection of student records that contain fields such as student name, major, date of birth, and social security number (SS#). The key field is the social security number, which is a nine digit decimal number. To keep the example manageable, assume that we have a total of five elements. Figure 12.8 shows the name and SS# fields for each of these five elements.

Name	SS#
Jack	951-94-1654
Jill	562-44-2169
Bill	271-16-3634
Kathy	278-49-1515
April	951-23-7625

Figure 12.8: Five elements (student records)

To obtain a trie representation for these five elements, we first select a radix that will be used to decompose each key into digits. If we use the radix 10, the

decomposed digits are just the decimal digits shown in Figure 12.8. We shall examine the digits of the key field (i.e., SS#) from left to right. Using the first digit of the SS#, we partition the elements into three groups—elements whose SS# begins with 2 (i.e., Bill and Kathy), those that begin with 5 (i.e., Jill), and those that begin with 9 (i.e., April and Jack). Groups with more than one element are partitioned using the next digit in the key. This partitioning process is continued until every group has exactly one element in it.

The partitioning process described above naturally results in a tree structure that has 10-way branching as is shown in Figure 12.9. The tree employs two types of nodes—*branch nodes* and *element nodes*. Each branch node has 10 children (or pointer) fields. These fields, `child[0:9]`, have been labeled 0, 1, ..., 9 for the root node of Figure 12.9. `root.child[0]` points to the root of a subtree that contains all elements whose first digit is 0. In Figure 12.9, nodes A, B, D, E, F, and J are branch nodes. The remaining nodes, nodes C, G, H, I, and K are element nodes. Each element node contains exactly one element. In Figure 12.9, only the key field of each element is shown in the element nodes.

12.3.2 Searching a Trie

Searching a trie for an element whose key is k requires breaking up k into its constituent characters/digits and following the branches determined by these characters. The function `Trie::Search` (Program 12.3) assumes that $p = \text{NULL}$ is not a branch node and that the function `digit(k,i)` returns the i th digit of k .

Analysis of `Trie::Search`: The search algorithm for tries is very straightforward, and one may readily verify that the worst-case search time is $O(l)$, where l is the number of levels in the trie (including both branch and element nodes).

In the case of an index, all trie nodes will reside on disk, so at most l disk accesses will be made during search. When the nodes reside on disk, the C++ pointer type cannot be used, as C++ does not allow input/output of pointers. The link data member will now be implemented as an integer. \square

12.3.3 Sampling Strategies

Given a set of key values to be represented in an index, the number of levels in the trie will depend on the strategy or key sampling technique used to determine the branching at each level. This can be defined by a sampling function, the `sample(x,i)`, which appropriately samples x for branching at the i th level. In the trie of Figure 12.6 and in the search algorithm `Trie::Search` (Program 12.3) this function is `sample(x,i) = i`th character of x . Some other choices for this function are

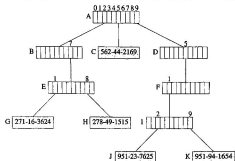


Figure 12.9: Trie for the elements of Figure 12.8

- (1) $\text{sample}(x, i) = x_{2^{i-1}+1}$
- (2) $\text{sample}(x, i) = x_{r(i)+1}$ for $r(i)$ a randomization function
- (3) $\text{sample}(x, i) = \begin{cases} x_{i/2} & \text{if } i \text{ is even} \\ x_{(i-1)/2} & \text{if } i \text{ is odd} \end{cases}$

where $x = x_1x_2 \cdots x_n$.

For each of these functions, one may easily construct key value sets for which the particular function is best (i.e., it results in a trie with the fewest number of levels). The trie of Figure 12.6 has five levels. Using function (1) on the same key values yields the trie of Figure 12.10, which has only three levels. An optimal sampling function for this data set will yield a trie that has only two levels (Figure 12.11). Choosing the optimal sampling function for any particular set of values is very difficult. In a dynamic situation, with insertion and deletion, we wish to optimize average performance. In the absence of any further

```

template <class K, E>
E* Trie<K, E>::Search (const K& k) const
// Search a trie. Return a pointer to the element whose key is k.
// Return NULL if no such element
    TrieNode<K, E> *p = root;
    for(int i = 1; p is a branch node; i++)
        p = p->child[ digit(k,i) ];
    if (p == NULL || p->key != k) return NULL;
    else return &p->element;
}
    
```

Program 12.3: Searching a trie

information on key values, the best choice would probably be function (2).



Figure 12.10: Trie constructed for data of Figure 12.6 sampling one character at a time, from right to left

Although all our examples of sampling have involved single-character sampling we need not restrict ourselves to this. The key value may be interpreted as consisting of digits using any radix we desire. Using a radix of 27^2 would result in two-character sampling. Other radices would give different samplings.

The maximum number of levels in a trie can be kept low by adopting a different strategy for element nodes. These nodes can be designed to hold more

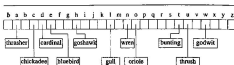


Figure 12.11: An optimal trie for the data of Figure 12.6 sampling on the first level done by using the fourth character of the key values

than one key value. If the maximum number of levels allowed is l , then all key values that are synonyms up to level $l-1$ are entered into the same element node. If the sampling function is chosen correctly, there will be only a few synonyms in each element node. The element node will therefore be small and can be processed in internal memory. Figure 12.12 shows the use of this strategy on the trie of Figure 12.6 with $l=3$. In further discussion we shall, for simplicity, assume that the sampling function in use is $\text{sample}(x,i) = \text{8th character of } x$ and that no restriction is placed on the number of levels in the trie.

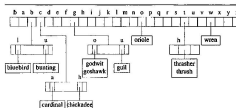


Figure 12.12: Trie obtained for data of Figure 12.6 when number of levels is limited to 3; keys have been sampled from left to right, one character at a time

12.3.4 Insertion into a Trie

Insertion into a trie is straightforward. We shall illustrate the procedure by two examples, and leave the formal writing of the algorithm as an exercise. Let us consider the trie of Figure 12.6 and insert into it the keys *bobwhite* and *bluejay*. First, we have $x = \text{bobwhite}$ and we attempt to search for *bobwhite*. This leads us to node α , where we discover that $\alpha.\text{link}[\alpha'] = 0$. Hence, x is not in the trie and may be inserted here (see Figure 12.13). Next, $x = \text{bluejay}$, and a search of the trie leads us to the element node that contains *bluebird*. The keys *bluebird* and *bluejay* are sampled until the sampling results in two different values. This happens at the fifth letter. Figure 12.13 shows the trie after both insertions.

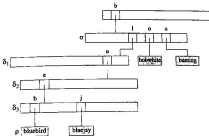


Figure 12.13: Section of trie of Figure 12.6 showing changes resulting from inserting *bobwhite* and *bluejay*

12.3.5 Deletion from a Trie

Once again, we shall not present the deletion algorithm formally but will look at two examples to illustrate some of the ideas involved in deleting entries from a

trie. From the trie of Figure 12.13, let us first delete *bobwhite*. To do this we set $\alpha.\text{link}['o']$ equal to 0. No other changes need to be made. Next, let us delete *bluejay*. This deletion leaves us with only one key value in the subtree, δ_3 . This means that the node δ_3 may be deleted, and p can be moved up one level. The same can be done for nodes δ_1 and δ_2 . Finally, the node α is reached. The subtree with root α has more than one key value. Therefore, p cannot be moved up any more levels, and we set $\alpha.\text{link}['r']$ equal to p . To facilitate deletions from tries, it is useful to add a *count* data member in each branch node. This data member contains the number of children the node has.

As in the case of binary tries, we can define compressed tries in which each branch node has at least two children. In this case, each branch node is augmented to have an additional data member, *skip*, that indicates the number of levels of branching that have been eliminated (alternately, we can have a data member, *sample*, that indicates the sampling level to use).

12.3.6 Keys With Different Length

As noted earlier, the keys in a trie must be such that no key is a prefix of another. When all keys are of the same length, as is the case in our SS# example of Figure 12.9, this property is assured. But, when keys are of different length, as is the case with the keys of Figure 12.6, it is possible for one key to be a prefix of another. A popular way to handle such key collections is to append a special character such as a blank or a # that doesn't appear in any key to the end of each key. This assures us that the modified keys (with the special character appended) satisfy the no-prefix property.

An alternative to adding a special character at the end of each key is to give each node a *data* field that is used to store the element (if any) whose key exhausts at that node. So, for example, the element whose key is 27 can be stored in node *E* of Figure 12.9. When this alternative is used, the search strategy is modified so that when the digits of the search key are exhausted, we examine the *data* field of the reached node. If this *data* field is empty, we have no element whose key equals the search key. Otherwise, the desired element is in this *data* field.

It is important to note that in applications that have different length keys with the property that no key is a prefix of another, neither of the just-mentioned strategies is needed.

12.3.7 Height of a Trie

In the worst case, a root-node to element-node path has a branch node for every digit in a key. Therefore, the height of a trie is at most $\text{numberofdigits} + 1$.

A trie for social security numbers has a height that is at most 10. If we assume that it takes the same time to move down one level of a trie as it does to move down one level of a binary search tree, then with at most 10 moves we can search a social-security trie. With this many moves, we can search a binary search tree that has at most $2^{10} - 1 = 1023$ elements. This means that, we expect searches in the social security trie to be faster than searches in a binary search tree (for student records) whenever the number of student records is more than 1023. The breakeven point will actually be less than 1023 because we will normally not be able to construct full or complete binary search trees for our element collection.

Since a SS# is nine digits, a social security trie can have up to 10^9 elements in it. An AVL tree with 10^9 elements can have a height that is as much as (approximately) $1.44 \log_2(10^9 + 2) = 44$. Therefore, it could take us four times as much time to search for elements when we organize our collection of student records as an AVL tree rather than as a trie!

12.3.8 Space Required and Alternative Node Structures

The use of branch nodes that have as many child fields as the radix of the digits (or one more than this radix when different keys may have different length) results in a fast search algorithm. However, this node structure is often wasteful of space because many of the child fields are NULL. A radix r trie for d digit keys requires $O(nh)$ child fields, where n is the number of elements in the trie. To see this, notice that in a d digit trie with n element nodes, each element node may have at most d ancestors, each of which is a branch node. Therefore, the number of branch nodes is at most dn . (Actually, we cannot have this many branch nodes, because the element nodes have common ancestors like the root node.)

We can reduce the space requirements, at the expense of increased search time, by changing the node structure. Some of the possible alternative structures for the branch node of a trie we considered below.

A chain of nodes.

Each node of the chain has the three fields *digitValue*, *child*, and *next*. Node *E* of Figure 12.9, for example, would be replaced by the chain shown in Figure 12.14.

The space required by a branch node changes from that required for r children/pointer fields to that required for $2p$ pointer fields and p digit value

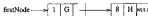


Figure 12.14: Chain for node E of Figure 12.9

fields, where p is the number of children fields in the branch node that are not NULL. Under the assumption that pointer fields and digit value fields are of the same size, a reduction in space is realized when more than two-thirds of the children fields in branch nodes are NULL. In the worst case, almost all the branch nodes have only 1 field that is not NULL and the space savings become almost $(1 - 3/4) = 100\%$.

A (balanced) binary search tree.

Each node of the binary search tree has a digit value and a pointer to the subtree for that digit value. Figure 12.15 shows the binary search tree for node E of Figure 12.9.



Figure 12.15: Binary search tree for node E of Figure 12.9

Under the assumption that digit values and pointers take the same amount of space, the binary search tree representation requires space for $4p$ fields per branch node, because each search tree node has fields for a digit value, a subtree pointer, a left child pointer, and a right child pointer. The binary search tree representation of a branch node saves us space when more than three-fourths of the children fields in branch nodes are NULL. Note that for large r , the binary search tree is faster to search than the chain described above.

A binary trie.

Figure 12.16 shows the binary trie for node E of Figure 12.9. The space required

by a branch node represented as a binary trie is at most $(2 \cdot \lceil \log_2 r \rceil + 1)p$.



Figure 12.16: Binary trie for node *E* of Figure 12.9

A hash table.

When a hash table with a sufficiently small loading density is used, the expected time performance is about the same as when the node structure of Figure 12.9 is used. Since we expect the fraction of NULL child fields in a branch node to vary from node to node and also to increase as we go down the trie, maximum space efficiency is obtained by consolidating all of the branch nodes into a single hash table. To accomplish this, each node in the trie is assigned a number, and each parent-to-child pointer is replaced by a tuple of the form $(currentNode, digitValue, childNode)$. The numbering scheme for nodes is chosen so as to easily distinguish between branch and element nodes. For example, if we expect to have at most 100 elements in the trie at any time, the numbers 0 through 99 are reserved for element nodes and the numbers 100 on up are used for branch nodes. The element nodes are themselves represented as an array *element*[100]. (An alternative scheme is to represent pointers as tuples of the form $(currentNode, digitValue, childNode, childNodeIsBranchNode)$, where *childNodeIsBranchNode* = true iff the child is a branch node.)

Suppose that the nodes of the trie of Figure 12.9 are assigned numbers as given in Figure 12.17. This number assignment assumes that the trie will have no more than 10 elements.

The pointers in node *A* are represented by the tuples $(10, 2, 11)$, $(10, 5, 0)$, and $(10, 9, 12)$. The pointers in node *E* are represented by the tuples $(13, 1, 1)$ and $(13, 8, 2)$.

The pointer tuples are stored in a hash table using the first two fields (i.e., the *currentNode* and *digitValue*) as the key. For this purpose, we may transform the two-field key into an integer using the formula $currentNode * r + digitValue$, where *r* is the trie radix, and use the division method to hash the transformed key into a home bucket. The data presently in element node *i* is stored in *element*[*i*].

Node	A	B	C	D	E	F	G	H	I	J	K
Number	10	11	0	12	13	14	1	2	15	3	4

Figure 12.17: Number assignment to nodes of trie of Figure 12.9

To see how all this works, suppose we have set up the trie of Figure 12.9 using the hash table scheme just described. Consider searching for an element with key 278-49-1513. We begin with the knowledge that the root node is assigned the number 10. Since the first digit of the search key is 2, we query our hash table for a pointer triple with key (10,2). The hash table search is successful and the triple (10,2,11) is retrieved. The *childNode* component of this triple is 11, and since all element nodes have a number 9 or less, the child node is determined to be a branch node. We make a move to the branch node 11. To move to the next level of the trie, we use the second digit 7 of the search key. For this move, we query the hash table for a pointer with key (11,7). Once again, the search is successful and the triple (11,7,13) is retrieved. The next query to the hash table is for a triple with key (13,8). This time, we obtain the triple (13,8,2). Since, *childNode* = 2 < 10, we know that the pointer gets us to an element node. So, we compare the search key with the key of *element* [2]. The keys match, and we have found the element we were looking for.

When searching for an element with key 323-16-8996, the first query is for a triple with key (10,3). The hash table has no triple with this key, and we conclude that the trie has no element whose key equals the search key.

The space needed for each pointer triple is about the same as that needed for each node in the chain of nodes representation of a trie node. Therefore, if we use a linear open addressed hash table with a loading density of α , the hash table scheme will take approximately $(1/\alpha - 1) \times 100\%$ more space than required by the chain of nodes scheme. However, when the hash table scheme is used, we can retrieve a pointer in $O(1)$ expected time, whereas the time to retrieve a pointer using the chain of nodes scheme is $O(r)$. When the (balanced) binary search tree or binary trie schemes are used, it takes $O(\log r)$ time to retrieve a pointer. For large radices, the hash table scheme provides significant space saving over the scheme of Figure 12.9 and results in a small constant factor degradation in the expected time required to perform a search.

The hash table scheme actually reduces the expected time to insert elements into a trie, because when the node structure of Figure 12.9 is used, we must spend $O(r)$ time to initialize each new branch node (see the description of the insert operation below). However, when a hash table is used, the insertion time is independent of the trie radix.

To support the removal of elements from a trie represented as a hash table, we must be able to reuse element nodes. This reuse is accomplished by setting up an available space list of element nodes that are currently not in use.

12.3.9 Prefix Search and Applications

You have probably realized that to search a trie we do not need the entire key. Most of the time, only the first few digits (i.e., a prefix) of the key is needed. For example, our search of the trie of Figure 12.9 for an element with key 951-23-7625 used only the first four digits of the key. The ability to search a trie using only the prefix of a key enables us to use tries in applications where only the prefix might be known or where we might desire the user to provide only a prefix. Some of these applications are described below.

Criminology: Suppose that you are at the scene of a crime and observe the first few characters *CRX* on the registration plate of the getaway car. If we have a trie of registration numbers, we can use the characters *CRX* to reach a subtree that contains all registration numbers that begin with *CRX*. The elements in this subtree can then be examined to see which cars satisfy other properties that might have been observed.

Automatic Command Completion: When using an operating system such as Unix or Windows (command prompt), we type in system commands to accomplish certain tasks. For example, the Unix and DOS command *cd* may be used to change the current directory. Figure 12.18 gives a list of commands that have the prefix *ps* (this list was obtained by executing the command *ls /usr/local/bin/ps** on a Unix system).

ps2ascii	ps2pdf	psbook	psmandup	psselect
ps2epsi	ps2pk	pscol	psmerge	psrpsrm
ps2frag	ps2ps	psdiaggm	psnp	psrps
ps2gif	psbb	pslatex	psresize	psrtract

Figure 12.18: Commands that begin with "ps"

We can simplify the task of typing in commands by providing a command completion facility which automatically types in the command suffix once the user has typed in a long enough prefix to uniquely identify the command. For

instance, once the letters *psi* have been entered, we know that the command must be *psidropgr* because there is only one command that has the prefix *psi*. In this case, we replace the need to type in a 9 character command name by the need to type in just the first 3 characters of the command!

A command completion system is easily implemented when the commands are stored in a trie using ASCII characters as the digits. As the user types the command digits from left to right, we move down the trie. The command may be completed as soon as we reach an element node. If we fall off the trie in the process, the user can be informed that no command with the typed prefix exists.

Although we have described command completion in the context of operating system commands, the facility is useful in other environments:

- (1) A web browser keeps a history of the URLs of sites that you have visited. By organizing this history as a trie, the user need only type the prefix of a previously used URL and the browser can complete the URL.
- (2) A word processor can maintain a collection of words and can complete words as you type the text. Words can be completed as soon as you have typed a long enough prefix to identify the word uniquely.
- (3) An automatic phone dialler can maintain a list of frequently called telephone numbers as a trie. Once you have punched in a long enough prefix to uniquely identify the phone number, the dialler can complete the call for you.

12.3.10 Compressed Tries

Take a close look at the trie of Figure 12.9. This trie has a few branch nodes (nodes *B*, *D*, and *F*) that do not partition the elements in their subtree into two or more nonempty groups. We often can improve both the time and space performance metrics of a trie by eliminating all branch nodes that have only one child. The resulting trie is called a *compressed trie*.

When branch nodes with a single child are removed from a trie, we need to keep additional information so that trie operations may be performed correctly. The additional information stored in these compressed trie structures is described below.

12.3.10.1 Compressed Tries with Digit Numbers

In a *compressed trie with digit numbers*, each branch node has an additional field *digitNumber* that tells us which digit of the key is used to branch at this node. Figure 12.19 shows the compressed trie with digit numbers that corresponds to the trie of Figure 12.9. The leftmost field of each branch node of Figure 12.19 is the *digitNumber* field.



Figure 12.19: Compressed trie with digit numbers

12.3.10.2 Searching a Compressed Trie with Digit Numbers

A compressed trie with digit numbers may be searched by following a path from the root. At each branch node, the digit, of the search key, given in the branch node's *digitNumber* field is used to determine which subtree to move to. For example, when searching the trie of Figure 12.19 for an element with key 951-23-7625, we start at the root of the trie. Since the root node is a branch node with *digitNumber*=1, we use the first digit 9 of the search key to determine which subtree to move to. A move to node $A.child[9]=I$ is made. Since, $I.digitNumber=4$, the fourth digit, 2, of the search key tells us which subtree to move to. A move is now made to node $I.child[2]=J$. We are now at an element node, and the search key is compared with the key of the element in node J . Since the keys match, we have found the desired element.

Notice that a search for an element with key 913-23-7625 also terminates

at node J . However, the search key and the element key at node J do not match and we conclude that the trie contains no element with key 913-23-7625.

12.3.10.3 Inserting into a Compressed Trie with Digit Numbers

To insert an element with key 987-26-1615 into the trie of Figure 12.19, we first search for an element with this key. The search ends at node J . Since, the search key and the key, 951-23-7625, of the element in this node do not match, we conclude that the trie has no element whose key matches the search key. To insert the new element, we find the first digit where the search key differs from the key in node J and create a branch node for this digit. Since, the first digit where the search key 987-26-1615 and the element key 951-23-7625 differ is the second digit, we create a branch node with `digitNumber = 2`. Since, digit values increase as we go down the trie, the proper place to insert the new branch node can be determined by retracing the path from the root to node J and stopping as soon as either a node with digit value greater than 2 or the node J is reached. In the trie of Figure 12.19, this path retracing stops at node I . The new branch node is made the parent of node J , and we get the trie of Figure 12.20.

Consider inserting an element with key 958-36-4194 into the compressed trie of Figure 12.19. The search for an element with this key terminates when we fall off the trie by following the pointer `lchild[3]=NULL`. To complete the insertion, we must first find an element in the subtree rooted at node I . This element is found by following a downward path from node I using (say) the first non NULL link in each branch node encountered. Doing this on the compressed trie of Figure 12.19, leads us to node J . Having reached an element node, we find the first digit where the element key and the search key differ and complete the insertion as in the previous example. Figure 12.21 shows the resulting compressed trie.

Because of the possible need to search for the first non NULL child pointer in each branch node, the time required to insert an element into a compressed tries with digit numbers is $O(ed)$, where e is the trie radix and d is the maximum number of digits in any key.

12.3.10.4 Deleting an Element from a Compressed Trie with Digit Numbers

To delete an element whose key is k , we do the following:

- (1) Find the element node X that contains the element whose key is k .
- (2) Discard node X .

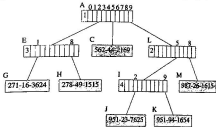


Figure 12.20: Compressed trie following the insertion of 987-26-1615 into the compressed trie of Figure 12.19

- (3) If the parent of X is left with only one child, discard the parent node also. When the parent of X is discarded, the sole remaining child of the parent of X becomes a child of the grandparent (if any) of X .

To remove the element with key 951-94-1654 from the compressed trie of Figure 12.21, we first locate the node K that contains the element that is to be removed. When this node is discarded, the parent I of K is left with only one child. Consequently, node I is also discarded, and the only remaining child J of node I is made a child of the grandparent of K . Figure 12.22 shows the resulting compressed trie.

Because of the need to determine whether a branch node is left with two or more children, removing a d -digit element from a radix r trie takes $O(d \cdot r)$ time.



Figure 12.21: Compressed trie following the insertion of 958-36-4194 into the compressed trie of Figure 12.19

12.3.11 Compressed Tries with Skip Fields

In a *compressed trie with skip fields*, each branch node has an additional field *skip* which tells us the number of branch nodes that were originally between the current branch node and its parent. Figure 12.23 shows the compressed trie with skip fields that corresponds to the trie of Figure 12.9. The leftmost field of each branch node of Figure 12.23 is the skip field.

The algorithms to search, insert, and remove are very similar to those used for a compressed trie with digit numbers.

12.3.12 Compressed Tries with Labeled Edges

In a *compressed trie with labeled edges*, each branch node has the following additional information associated with it: a pointer/reference *element* to an element (or element node) in the subtrie, and an integer *skip* which equals the number of branch nodes eliminated between this branch node and its parent. Figure 12.24 shows the compressed trie with labeled edges that corresponds to

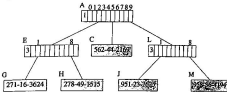


Figure 12.22: Compressed trie following the removal of 951-94-1654 from the compressed trie of Figure 12.21



Figure 12.23: Compressed trie with skip fields

the trie of Figure 12.9. The first field of each branch node is its *element* field, and the second field is the *skip* field.

Even though we store the “label” with branch nodes, it is convenient to think of this information as being associated with the edge that comes into the

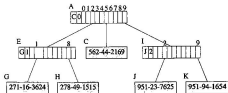


Figure 12.24: Compressed trie with labeled edges

branch node from its parent (when the branch node is not the root). When moving down a trie, we follow edges, and when an edge is followed, we skip over the number of digits given by the skip field of the edge information. The value of the digits that are skipped over may be determined by using the *element* field.

When moving from node *A* to node *I* of the compressed trie of Figure 12.24, we use digit 1 of the key to determine which child field of *A* is to be used. Also, we skip over the next 2 digits, that is, digits 2 and 3, of the keys of the elements in the subtree rooted at *I*. Since all elements in the subtree *I* have the same value for the digits that are skipped over, we can determine the value of these skipped over digits from any of the elements in the subtree. Using the *element* field of the edge label, we access the element node *J*, and determine that the digits that are skipped over are 5 and 1.

12.3.12.1 Searching a Compressed Trie with Labeled Edges

When searching a compressed trie with labeled edges, we can use the edge label to terminate unsuccessful searches (possibly) before we reach an element node or fall off the trie. As in the other compressed trie variants, the search is done by following a path from the root. Suppose we are searching the compressed trie of Figure 12.24 for an element with key 931-23-1234. Since the skip value for the root node is 0, we use the first digit 9 of the search key to determine which subtree to move to. A move to node $A.child[9]=I$ is made. By examining the edge label (stored in node *I*), we determine that, in making the move from node *A* to

node *I*, the digits 5 and 1 are skipped. Since these digits do not agree with the next two digits of the search key, the search terminates with the conclusion that the trie contains no element whose key equals the search key.

12.3.12.2 Inserting into a Compressed Trie with Labeled Edges

To insert an element with key 987-26-1615 into the compressed trie of Figure 12.24, we first search for an element with this key. The search terminates unsuccessfully when we move from node *A* to node *I* because of a mismatch between the skipped over digits and the corresponding digits of the search key. The first mismatch is at the first skipped over digit. Therefore, we insert a branch node *L*. The skip value for this branch node is 0, and its *element* field is set to reference the element node for the newly inserted element. We must also change the skip value of *I* to 1. Figure 12.25 shows the resulting compressed trie.



Figure 12.25: Compressed trie following the insertion of 987-26-1615 into the compressed trie of Figure 12.24

Suppose we are to insert an element with key 958-36-4194 into the compressed trie of Figure 12.25. The search for an element with this key

terminates when we move to node *I* because of a mismatch between the digits that are skipped over and the corresponding digits of the search key. A new branch node is inserted between nodes *A* and *I* and we get the compressed trie that is shown in Figure 12.26.



Figure 12.26: Compressed trie following the insertion of 958-36-4194 into the compressed trie of Figure 12.24

The time required to insert a d digit element into a radix r compressed trie with labeled edges is $O(r \cdot d)$.

12.3.12.3 Deleting an Element from a Compressed Trie with Labeled Edges

This is similar to removal from a compressed trie with digit numbers except for the need to update the *element* fields of branch nodes whose *element* field references the removed element.

12.3.13 Space Required by a Compressed Trie

Since each branch node partitions the elements in its subtree into two or more nonempty groups, an n element compressed trie has at most $n-1$ branch nodes. Therefore, the space required by each of the compressed trie variants described by us is $O(nr)$, where r is the trie radix.

When compressed tries are represented as hash tables, we need an additional data structure to store the nonpointer fields of branch nodes. We may use an array for this purpose.

EXERCISES

- Draw the trie obtained for the following data:
 AMIOT, AVENGER, AVRO, HEINKEL, HELLDIVER, MACCHI,
 MARAUDER, MUSTANG, SPITFIRE, SYKHOI
 Sample the keys from left to right one character at a time.
 - Using single-character sampling, obtain a trie with the fewest number of levels.
- Explain how a trie could be used to implement a spelling checker.
- Explain how a trie could be used to implement an auto-command completion program. Such a program would maintain a library of valid commands. It would then accept a user command, character by character, from a keyboard. When a sufficient number of characters had been input to uniquely identify the command, it would display the complete command on the computer monitor.
- Write an algorithm to insert a key value x into a trie in which the keys are sampled from left to right, one character at a time.
- Do Exercise 4 with the added assumption that the trie is to have no more than six levels. Synonyms are to be packed into the same element node.
- Write an algorithm to delete x from a trie under the assumptions of Exercise 4. Assume that each branch node has a cover data member equal to the number of element nodes in the subtree for which it is the root.
- Do Exercise 6 for the trie of Exercise 5.
- In the trie of Figure 12.13 the nodes δ_1 and δ_2 each have only one child. Branch nodes with only one child may be eliminated from tries by maintaining a *skip* data member with each node. The value of this data member equals the number of characters to be skipped before obtaining the next character to be sampled. Thus, we can have $skip[\delta_1] = 2$ and delete the

- nodes δ_1 and δ_2 . Write algorithms to search, insert, and delete from tries in which each branch node has a skip data member.
9. Assume that the branch nodes of a compressed trie are represented using a hash table (one for each node). Each such hash table is augmented with a count and skip value as described above. Describe how this change to the node structure affects the time and space complexity of the trie data structure.
 10. Do the previous exercise for the case when each branch node is represented by a chain in which each node has two data members: *pointer* and *link*, where *pointer* points to a subtrie and *link* points to the next node in the chain. The number of nodes in the chain for any branch node equals the number of non-0 pointers in that node. Each chain is augmented by a skip value. Draw the chain representation of the compressed version of the trie of Figure 12.6.

12.4 SUFFIX TREES

12.4.1 Have You Seen This String?

In the classical *substring search* problem, we are given a string S and a pattern P and are to report whether or not the pattern P occurs in the string S . For example, the pattern $P = \text{cat}$ appears (twice) in the string $S1 = \text{The big cat ate the small catfish}$, but does not appear in the string $S2 = \text{Dogs for sale}$.

Researchers in the human genome project, for example, are constantly searching for substrings/patterns (we use the terms substring and pattern interchangeably) in a gene databank that contains tens of thousands of genes. Each gene is represented as a sequence or string of letters drawn from the alphabet A, C, G, T . Although most of the strings in the databank are around 2000 letters long, some have tens of thousands of letters. Because of the size of the gene databank and the frequency with which substring searches are done, it is imperative that we have as fast an algorithm as possible to locate a given substring within the strings in the databank.

We can search for a pattern P in a string S using Program 2.16. The complexity of such a search is $O(|P| + |S|)$, where $|P|$ denotes the length (i.e., number of letters or digits) of P . This complexity looks pretty good when you consider that the pattern P could appear anywhere in the string S . Therefore, we must examine every letter/digit (we use the terms letter and digit interchangeably) of the string before we can conclude that the search pattern does not appear in the string. Further, before we can conclude that the search pattern appears in the string, we must examine every digit of the pattern. Hence, every pattern search algorithm must take time that is linear in the lengths of the pattern and the

string being searched.

When classical pattern matching algorithms are used to search for several patterns P_1, P_2, \dots, P_k in the string S , $O(|P_1| + |P_2| + \dots + |P_k| + k|S|)$ time is taken (because $O(|P_i| + |S|)$ time is taken to search for P_i). The suffix tree data structure that we are about to study reduces this complexity to $O(|P_1| + |P_2| + \dots + |P_k| + |S|)$. Of this time, $O(|S|)$ time is spent setting up the suffix tree for the string S ; an individual pattern search takes only $O(|P_i|)$ time (after the suffix tree for S has been built). Therefore once the suffix tree for S has been created, the time needed to search for a pattern depends only on the length of the pattern.

12.4.2 The Suffix Tree Data Structure

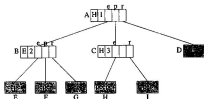
The *suffix tree* for S is actually the compressed trie for the nonempty suffixes of the string S . Since a suffix tree is a compressed trie, we sometimes refer to the tree as a trie and to its subtrees as subtries.

The (nonempty) suffixes of the string $S = \text{peeper}$ are *peeper*, *eeper*, *eper*, *per*, *er*, and *r*. Therefore, the suffix tree for the string *peeper* is the compressed trie that contains the elements (which are also the keys) *peeper*, *eeper*, *eper*, *per*, *er*, and *r*. The alphabet for the string *peeper* is *e,p,r*. Therefore, the radix of the compressed trie is 3. If necessary, we may use the mapping $e \rightarrow 0$, $p \rightarrow 1$, $r \rightarrow 2$, to convert from the letters of the string to numbers. This conversion is necessary only when we use a node structure in which each node has an array of child pointers. Figure 12.27 shows the compressed trie (with labeled edges) for the suffixes of *peeper*. This compressed trie is also the suffix tree for the string *peeper*.

Since the data in the element nodes $D[0]$ are the suffixes of *peeper*, each element node need retain only the start index of the suffix it contains. When the letters in *peeper* are indexed from left to right beginning with the index 1, the element nodes $D[0]$ need only retain the indexes 6, 2, 3, 5, 1, and 4, respectively. Using the index stored in an element node, we can access the suffix from the string S . Figure 12.28 shows the suffix tree of Figure 12.27 with each element node containing a suffix index.

The first component of each branch node is a reference to an element in that subtrie. We may replace the element reference by the index of the first digit of the referenced element. Figure 12.29 shows the resulting compressed trie. We shall use this modified form as the representation for the suffix tree.

When describing the search and construction algorithms for suffix trees, it is easier to deal with a drawing of the suffix tree in which the edges are labeled by the digits used in the move from a branch node to a child node. The first digit of the label is the digit used to determine which child is moved to, and the

Figure 12.27: Compressed trie for the suffixes of *peeper*

remaining digits of the label give the digits that are skipped over. Figure 12.30 shows the suffix tree of Figure 12.29 drawn in this manner.

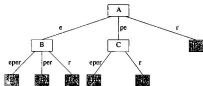
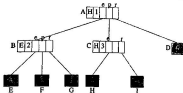


Figure 12.30: A more humane drawing of a suffix tree

In the more humane drawing of a suffix tree, the labels on the edges on any root to element node path spell out the suffix represented by that element node. When the digit number for the root is not 1, the humane drawing of a suffix tree includes a header node with an edge to the former root. This edge is labeled with



$$S = \begin{array}{|c|c|c|c|c|c|} \hline 1 & 2 & 3 & 4 & 5 & 6 \\ \hline p & e & p & p & e & r \\ \hline \end{array}$$

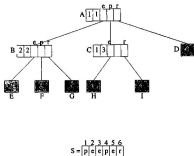
 Figure 12.28: Modified compressed trie for the suffixes of *pepper*

the digits that are skipped over.

The string represented by a node of a suffix tree is the string formed by the labels on the path from the root to that node. Node *A* of Figure 12.30 represents the empty string ϵ , node *C* represents the string *pe*, and node *F* represents the string *eper*.

Since the keys in a suffix tree are of different length, we must ensure that no key is a proper prefix of another. Whenever the last digit of string *S* appears only once in *S*, no suffix of *S* can be a proper prefix of another suffix of *S*. In the string *pepper*, the last digit is *r*, and this digit appears only once. Therefore, no suffix of *pepper* is a proper prefix of another. The last digit of *data* is *a*, and this last digit appears twice in *data*. Therefore, *data* has two suffixes *ata* and *a* that begin with *a*. The suffix *a* is a proper prefix of the suffix *ata*.

When the last digit of the string *S* appears more than once in *S* we must append a new digit (say $\#$) to the suffixes of *S* so that no suffix is a prefix of another. Optionally, we may append the new digit to *S* to get the string *S#*, and then construct the suffix tree for *S#*. When this optional route is taken, the suffix

Figure 12.29: Suffix tree for *peeper*

tree has one more suffix (*r*) than the suffix tree obtained by appending the symbol *r* to the suffixes of *S*.

12.4.3 Let's Find That Substring (Searching a Suffix Tree)

But First, Some Terminology

Let $n = |S|$ denote the length (i.e., number of digits) of the string whose suffix tree we are to build. We number the digits of *S* from left to right beginning with the number 1. $S[i]$ denotes the *i*th digit of *S*, and $\text{suffix}(i)$ denotes the suffix $S[i] \cdots S[n]$ that begins at digit *i*, $1 \leq i \leq n$.

On With the Search

A fundamental observation used when searching for a pattern *P* in a string *S* is that *P* appears in *S* (i.e., *P* is a substring of *S*) iff *P* is a prefix of some suffix of *S*.

Suppose that $P = P[1] \cdots P[k] = S[i] \cdots S[i+k-1]$. Then, *P* is a prefix

of $\text{suffix}(i)$. Since $\text{suffix}(i)$ is in our compressed trie (i.e., suffix tree), we can search for P by using the strategy to search for a key prefix in a compressed trie.

Let's search for the pattern $P = \text{per}$ in the string $S = \text{peeper}$. Imagine that we have already constructed the suffix tree (Figure 12.30) for *peeper*. The search starts at the root node A . Since $P[1] = p$, we follow the edge whose label begins with the digit p . When following this edge, we compare the remaining digits of the edge label with successive digits of P . Since these remaining label digits agree with the pattern digits, we reach the branch node C . In going to node C , we have used the first two digits of the pattern. The third digit of the pattern is r , and so, from node C we follow the edge whose label begins with r . Since this edge has no additional digits in its label, no additional digit comparisons are done and we reach the element node I . At this time, the digits in the pattern have been exhausted and we conclude that the pattern is in the string. Since an element node is reached, we conclude that the pattern is actually a suffix of the string *peeper*. In the actual suffix tree representation (rather than in the human drawing), the element node I contains the index 4 which tells us that the pattern $P = \text{per}$ begins at digit 4 of *peeper* (i.e., $P = \text{suffix}(4)$). Further, we can conclude that *per* appears exactly once in *peeper*; the search for a pattern that appears more than once terminates at a branch node, not at an element node.

Now, let us search for the pattern $P = \text{enee}$. Again, we start at the root. Since the first character of the pattern is e , we follow the edge whose label begins with e and reach the node B . The next digit of the pattern is also e , and so, from node B we follow the edge whose label begins with e . In following this edge, we must compare the remaining digits *nee* of the edge label with the following digits *ee* of the pattern. We find a mismatch when the first pair (p, e) of digits are compared and we conclude that the pattern does not appear in *peeper*.

Suppose we are to search for the pattern $P = p$. From the root, we follow the edge whose label begins with p . In following this edge, we compare the remaining digits (only the digit e remains) of the edge label with the following digits (there aren't any) of the pattern. Since the pattern is exhausted while following this edge, we conclude that the pattern is a prefix of all keys in the subtree rooted at node C . We can find all occurrences of the pattern by traversing the subtree rooted at C and visiting the information nodes in this subtree. If we want the location of just one of the occurrences of the pattern, we can use the index stored in the first component of the branch node C (see Figure 12.29). When a pattern exhausts while following the edge to node X , we say that node X has been reached; the search terminates at node X .

When searching for the pattern $P = \text{rope}$, we use the first digit r of P and reach the element node D . Since the the pattern has not been exhausted, we must check the remaining digits of the pattern against those of the key in D . This check reveals that the pattern is not a prefix of the key in D , and so the pattern does not appear in *peeper*.

The last search we are going to do is for the pattern $P = \text{pepe}$. Starting at the root of Figure 12.30, we move over the edge whose label begins with p and reach node C . The next unexamined digit of the search pattern is e . So, from node C , we wish to follow the edge whose label begins with e . Since no edge satisfies this requirement, we conclude that pepe does not appear in the string peper .

12.4.4 Other Nifty Things You Can Do with a Suffix Tree

Once we have set up the suffix tree for a string S , we can tell whether or not S contains a pattern P in $O(|P|)$ time. This means that if we have a suffix tree for the text of Shakespeare's play "Romeo and Juliet," we can determine whether or not the phrase *wherefore art thou* appears in this play with lightning speed. In fact, the time taken will be that needed to compare up to 18 (the length of the search pattern) letters/digits. The search time is independent of the length of the play.

Some other interesting things you can do at lightning speed are described below.

Find all occurrences of a pattern P .

This is done by searching the suffix tree for P . If P appears at least once, the search terminates successfully either at an element node or at a branch node. When the search terminates at an element node, the pattern occurs exactly once. When we terminate at a branch node X , all places where the pattern occurs can be found by visiting the element nodes in the subtree rooted at X . This visiting can be done in time linear in the number of occurrences of the pattern if we

- (a) Link all of the element nodes in the suffix tree into a chain, the linking is done in lexicographic order of the represented suffixes (which also is the order in which the element nodes are encountered in a left to right scan of the element nodes). The element nodes of Figure 12.30 will be linked in the order E, F, G, H, I, D .
- (b) In each branch node, keep a reference to the first and last element node in the subtree of which that branch node is the root. In Figure 12.30, nodes A , B , and C keep the pairs (E, D) , (E, G) , and (H, I) , respectively. We use the pair $(firstInformationNode, lastInformationNode)$ to traverse the element node chain starting at $firstInformationNode$ and ending at $lastInformationNode$. This traversal yields all occurrences of patterns that begin with the string spelled by the edge labels from the root to the branch node. Notice that when $(firstInformationNode, lastInformationNode)$ pairs are kept in branch nodes, we can eliminate the branch node field that keeps a reference to an element node in the subtree (i.e., the field *element*).

Find all strings that contain a pattern P .

Suppose we have a collection S_1, S_2, \dots, S_k of strings and we wish to report all strings that contain a query pattern P . For example, the genome databank contains tens of thousands of strings, and when a researcher submits a query string, we are to report all databank strings that contain the query string. To answer queries of this type efficiently, we set up a compressed trie (we may call this a *multiple string suffix tree*) that contains the suffixes of the string S ($SS^{\#}1\$, SS^{\#}2\$, \dots, SS^{\#}k\)$, where S and $\#$ are two different digits that do not appear in any of the strings S_1, S_2, \dots, S_k). In each node of the suffix tree, we keep a list of all strings S_i that are the start point of a suffix, represented by an element node in that subtree.

Find the longest substring of S that appears at least $m > 1$ times.

This query can be answered in $O(|S|)$ time in the following way:

- (a) Traverse the suffix tree labeling the branch nodes with the sum of the label lengths from the root and also with the number of information nodes in the subtree.
- (b) Traverse the suffix tree visiting branch nodes with element node count $2m$. Determine the visited branch node with longest label length.

Note that step (a) needs to be done only once. Following this, we can do step (b) for as many values of m as is desired. Also, note that when $m = 2$ we can avoid determining the number of element nodes in subtrees. In a compressed trie, every subtree rooted at a branch node has at least two element nodes in it.

Find the longest common substring of the strings S and T .

This can be done in time $O(|S| + |T|)$ as below:

- (a) Construct a multiple string suffix tree for S and T (i.e., the suffix tree for $SS^{\#}T^{\#}$).
- (b) Traverse the suffix tree to identify the branch node for which the sum of the label lengths on the path from the root is maximum and whose subtree has at least one information node that represents a suffix that begins in S and at least one information node that represents a suffix that begins in T .

EXERCISES

1. Draw the suffix tree for $S = ababab\#$.
2. Draw the suffix tree for $S = aaaaaa\#$.
3. Draw the multiple string suffix tree for $S1 = abba$, $S2 = bbbb$, and $S3 = aaab$.

12.5 TRIES AND INTERNET PACKET FORWARDING

12.5.1 IP Routing

In the Internet, data packets are transported from source to destination by a series of routers. For example, a packet that originates in New York and is destined for Los Angeles will first be processed by a router in New York. This router may forward the packet to a router in Chicago, which, in turn, may forward the packet to a router in Denver. Finally, the router in Denver may forward the packet to Los Angeles. Each router moves a packet one step closer to its destination. A router does this by examining the destination address in the header of the packet to be routed. Using this destination address and a collection of forwarding rules stored in the router, the router decides where to send the packet next.

An Internet router table is a collection of rules of the form (P, NH) , where P is a prefix and NH is the next hop; NH is the next hop for packets whose destination address has the prefix P . For example, the rule $(01^*, a)$ states that the next hop for packets whose destination address (in binary) begins with 01 is a . In IPv4 (Internet Protocol version 4), destination addresses are 32 bits long. So, P may be up to 32 bits long. In IPv6, destination addresses are 128 bits long and so, P may be up to 128 bits in length.

It is not uncommon for a destination address to be matched by more than 1 rule in a commercial router table. In this case, the next hop is determined by the matching rule that has the longest prefix. So, for example, suppose that $(01^*, a)$ and $(0100^*, b)$ are the only two rules in our router table that match a packet whose destination address begins with the bit sequence 0100. The next hop for this packet is b . In other words, packet forwarding in the Internet is done by determining the longest matching prefix.

Although Internet router tables are dynamic in practice (i.e., the rule set changes in time; rules are added and deleted as routers come online and go offline), data structures for Internet router tables often are optimized for the search operation—given a destination address, determine the next hop for the longest matching prefix.

12.5.2 1-Bit Tries

A *1-bit trie* is very similar to a binary trie. It is a tree-like structure in which each node has a left child, left data, right child, and right data field. Nodes at level i of the trie store prefixes whose length is i . If the rightmost bit in a prefix whose length is i is 0, the prefix is stored in the left data field of a node that is at level i ; otherwise, the prefix is stored in the right data field of a node that is at level i . At level i of a trie, branching is done by examining bit i (bits are numbered from left to right beginning with the number 1) of a prefix or destination address. When bit i is 0, we move into the left subtree; when the bit is 1, we move into the right subtree. Figure 12.31(a) gives a set of 8 prefixes, and Figure 12.31(b) shows the corresponding 1-bit trie.

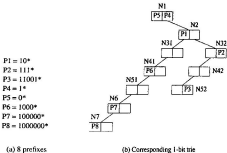


Figure 12.31: Prefixes and corresponding 1-bit trie

The height of a 1-bit trie is $O(W)$, where W is the length of the longest prefix in the router table. Note that $W \leq 32$ for IPv4 tables and $W \leq 128$ for IPv6 tables. Note also that there is no place, in a 1-bit trie, to store the prefix $*$ whose length is zero. This doesn't lead to any difficulty as this prefix matches every

destination address. In case a search of a 1-bit trie fails to find a matching prefix, the next-hop associated with * is used.

For any destination address d , all prefixes that match d lie on the search path determined by the bits of d . By following this search path, we may determine the longest matching-prefix in $O(W)$ time. Further, prefixes may be inserted/deleted in $O(W)$ time. The memory required by a 1-bit trie is $O(sW)$, where s is the number of rules in the router table.

Although the algorithms to search, insert and delete using a 1-bit trie are simple and of seemingly low complexity, $O(W)$, the demands of the Internet make the 1-bit trie impractical. Using trie-like structures, most of the time spent searching for the next hop goes to memory accesses. Hence, when analyzing the complexity of trie data structures for router tables, we focus on the number of memory accesses. When a 1-bit trie is used, it may take us up to W memory accesses to determine the next hop for a packet. Recall that $W \leq 32$ for IPv4 and $W \leq 128$ for IPv6. To keep the Internet operating smoothly, it is necessary that the next hop for each packet be determined using far fewer memory accesses than W . In practice, we must determine the next hop using at most (say) 6 memory accesses.

12.5.3 Fixed-Stride Tries

Since the trie of Figure 12.31(b) has a height of 7, a search into this trie may make up to 7 memory accesses, one access for each node on the path from the root to a node at level 7 of the trie. The total memory required for the 1-bit trie of Figure 12.31(b) is 20 units (each node requires 2 units, one for each pair of (child, data) fields). We may reduce the height of the router-table trie at the expense of increased memory requirement by increasing the branching factor at each node, that is, we use a multiway trie. The *stride* of a node is defined to be the number of bits used at that node to determine which branch to take. A node whose stride is s has 2^s child fields (corresponding to the 2^s possible values for the s bits that are used) and 2^s data fields. Such a node requires 2^s memory units. In a *fixed-stride trie* (FST), all nodes at the same level have the same stride; nodes at different levels may have different strides.

Suppose we wish to represent the prefixes of Figure 12.31(a) using an FST that has three levels. Assume that the strides are 2, 3, and 2. The root of the trie stores prefixes whose length is 2; the level two nodes store prefixes whose length is $5 (2 + 3)$; and level three nodes store prefixes whose length is $7 (2 + 3 + 2)$. This poses a problem for the prefixes of our example, because the length of some of these prefixes is different from the storable lengths. For instance, the length of P5 is 1. To get around this problem, a prefix with a nonpermissible length is expanded to the next permissible length. For example, $P5 = Q^*$ is expanded to

$P5a = 00^*$ and $P5b = 01^*$. If one of the newly created prefixes is a duplicate, natural dominance rules are used to eliminate all but one occurrence of the prefix. For instance, $P4 = 1^*$ is expanded to $P4a = 10^*$ and $P4b = 11^*$. However, $P1 = 10^*$ is to be chosen over $P4a = 10^*$, because $P1$ is a longer match than $P4$. So, $P4a$ is eliminated. Because of the elimination of duplicate prefixes from the expanded prefix set, all prefixes are distinct. Figure 12.32(a) shows the prefixes that result when we expand the prefixes of Figure 12.31 to lengths 2, 5, and 7. Figure 12.32(b) shows the corresponding FST whose height is 3 and whose strides are 2, 3, and 2.

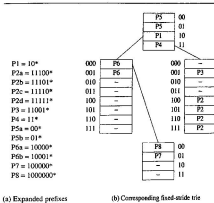


Figure 12.32: Prefix expansion and fixed-stride trie

Since the trie of Figure 12.32(b) can be searched with at most 3 memory accesses, it represents a time performance improvement over the 1-bit trie of

Figure 12.31(b), which requires up to 7 memory references to perform a search. However, the space requirements of the FST of Figure 12.32(b) are more than that of the corresponding 1-bit trie. For the root of the FST, we need 8 fields or 4 units; the two level 2 nodes require 8 units each; and the level 3 node requires 4 units. The total is 24 memory units.

We may represent the prefixes of Figure 12.31(a) using a one-level trie whose root has a stride of 7. Using such a trie, searches could be performed making a single memory access. However, the one-level trie would require $2^7 = 128$ memory units.

In the *fixed-stride trie optimization* (FSTO) problem, we are given a set P of prefixes and an integer k . We are to select the strides for a k -level FST in such a manner that the k -level FST for the given prefixes uses the smallest amount of memory.

For some P , a k -level FST may actually require more space than a $(k-1)$ -level FST. For example, when $P = \{00^*, 01^*, 10^*, 11^*\}$, the unique 1-level FST for P requires 4 memory units while the unique 2-level FST (which is actually the 1-bit trie for P) requires 6 memory units. Since the search time for a $(k-1)$ -level FST is less than that for a k -level tree, we would actually prefer $(k-1)$ -level FSTs that take less (or even equal) memory over k -level FSTs. Therefore, in practice, we are really interested in determining the best FST that uses at most k levels (rather than exactly k levels). The *modified MSTO problem* (MFSTO) is to determine the best FST that uses at most k levels for the given prefix set P .

Let O be the 1-bit trie for the given set of prefixes, and let F be any k -level FST for this prefix set. Let s_1, \dots, s_k be the strides for F . We shall say that level j , $1 \leq j \leq k$, of F covers levels a, \dots, b of O , where $a = \sum_{i=1}^{j-1} s_i + 1$ and $b = \sum_{i=1}^j s_i$. So, level 1 of the FST of Figure 12.32(b) covers levels 1 and 2 of the 1-bit trie of Figure 12.31(b). Level 2 of this FST covers levels 3, 4, and 5 of the 1-bit trie of Figure 12.31(b); and level 3 of this FST covers levels 6 and 7 of the 1-bit trie. We shall refer to levels $a, \dots, b = \sum_{i=1}^j s_i$, $1 \leq a \leq k$ as the *expansion levels* of O . The expansion levels defined by the FST of Figure 12.32(b) are 1, 3, and 6.

Let $noder(i)$ be the number of nodes at level i of the 1-bit trie O . For the 1-bit trie of Figure 12.31(a), $noder(1:7) = \{1, 1, 2, 2, 2, 1, 1\}$. The memory required by F is $\sum_{i=1}^k noder(s_i) \cdot 2^{s_i}$. For example, the memory required by the FST of Figure 12.32(b) is $noder(1) \cdot 2^1 + noder(3) \cdot 2^3 + noder(6) \cdot 2^3 = 24$.

Let $T(j, r)$ be the best (i.e., uses least memory) FST that uses at most r expansion levels and covers levels 1 through j of the 1-bit trie O . Let $C(j, r)$ be the cost (i.e., memory requirement) of $T(j, r)$. So, $T(W, k)$ is the best FST for O that uses at most k expansion levels and $C(W, k)$ is the cost of this FST. We observe that the last expansion level in $T(j, r)$ covers levels $m+1$ through j of O

for some m in the range 0 through $j - 1$ and the remaining levels of this best FST define $T(m, r - 1)$. So,

$$C(j, r) = \min_{0 \leq m < j} \{C(m, r - 1) + \text{nodes}(m + 1) * 2^{j-m-1}\}, j \geq 1, r > 1 \quad (12.1)$$

$$C(0, r) = 0 \text{ and } C(j, 1) = 2^j, j \geq 1 \quad (12.2)$$

Let $M(j, r)$, $r > 1$, be the smallest m that minimizes

$$C(m, r - 1) + \text{nodes}(m + 1) * 2^{j-m-1},$$

in Eq. 12.1. Eqs. 12.1 and 12.2 result in an algorithm to compute $C(W, k)$ in $O(kW^2)$. The $M(j, r)$ s may be computed in the same amount of time while we are computing the $C(j, r)$ s. Using the computed M values, the strides for the optimal FST that uses at most k expansion levels may be determined in an additional $O(k)$ time.

12.5.4 Variable-Stride Tries

In a *variable-stride trie* (VST) nodes at the same level may have different strides. Figure 12.33 shows a two-level VST for the 1-bit trie of Figure 12.31. The stride for the root is 2; that for the left child of the root is 5; and that for the root's right child is 3. The memory required by this VST is 4 (root) + 32 (left child of root) + 8 (right child of root) = 44.

Since PSTs are a special case of VSTs, the memory required by the best VST for a given prefix set P and number of expansion levels k is less than or equal to that required by the best PST for P and k .

Let r -VST be a VST that has at most r levels. Let $Opt(N, r)$ be the cost (i.e., memory requirement) of the best r -VST for a 1-bit trie whose root is N . The root of this best VST covers levels 1 through s of D for some s in the range 1 through $height(N)$ and the subtrees of this root must be best $(r - 1)$ -VSTs for the descendants of N that are at level $s + 1$ of the subtree rooted at N . So,

$$Opt(N, r) = \min_{1 \leq s \leq height(N)} \{2^s + \sum_{N' \in D_{s+1}(N)} Opt(N', r - 1)\}, r > 1 \quad (12.3)$$

where $D_s(N)$ is the set of all descendants of N that are at level s of N . For example, $D_1(N)$ is the set of children of N and $D_2(N)$ is the set of grandchildren of N . $height(N)$ is the maximum level at which the trie rooted at N has a node. For example, in Figure 12.31(b), the height of the trie rooted at $N1$ is 7. When $r = 1$,

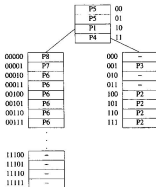


Figure 12.33: Two-level VST for prefixes of Figure 12.31(a)

$$Opt(N, 1) = 2^{\text{height}(N)}. \quad (12.4)$$

Let

$$Opt(N, s, r) = \sum_{M \in D(N)} Opt(M, r), \quad s \geq 1, \quad r \geq 1,$$

and let $Opt(N, 1, r) = Opt(N, r)$. From Eqs. 12.3 and 12.4, it follows that:

$$Opt(N, 1, r) = \min_{M \in D(N)} \{2^s + Opt(N, s+1, r-1)\}, \quad r \geq 1 \quad (12.5)$$

and

$$Opr(N, 1, 1) = 2^{\log_2 W}, \quad (12.6)$$

For $s > 1$ and $r > 1$, we get

$$\begin{aligned} Opr(N, s, r) &= \sum_{M \in D_s(N)} Opr(M, r) \\ &= Opr(LeftChild(N), s-1, r) \\ &\quad + Opr(RightChild(N), s-1, r). \end{aligned} \quad (12.7)$$

For Eq. 12.7, we need the following initial condition:

$$Opr(null, *, *) = 0 \quad (12.8)$$

For an n -rule router table, the 1-bit trie \mathcal{O} has $\mathcal{O}(nW)$ nodes. So, the number of $Opr(*, *, *)$ values is $\mathcal{O}(nW^2k)$. Each $Opr(*, s, *)$, $s > 1$, value may be computed in $\mathcal{O}(1)$ time using Eqs. 12.7 and 12.8 provided the Opr values are computed in postorder. The $Opr(*, 1, *)$ values may then be computed in $\mathcal{O}(W)$ time each using Eqs. 12.5 and 12.6. Therefore, we may compute $Opr(R, k) = Opr(R, 1, k)$, where R is the root of \mathcal{O} , in $\mathcal{O}(nW^2k)$ time. If we keep track of the s that minimizes the right side of Eq. 12.5 for each pair (N, r) , we can determine the strides of all nodes in the optimal k -VST in an additional $\mathcal{O}(nW)$ time.

EXERCISES

1. (a) Write a C++ function to compute $C(j, r)$ for $0 \leq j \leq W$ and $1 \leq r \leq k$ using Eqs. 12.1 and 12.2. Your function should compute $M(j, r)$ as well. The complexity of your function should be $\mathcal{O}(kW^2)$. Show that this is the case.
- (b) Write a C++ function that determines the strides of all levels in the best PST that has at most k levels. Your function should use the M values computed in part (a). The complexity of your function should be $\mathcal{O}(k)$. Show that this is the case.

2. (a) Write a C++ function to compute $Opr(N, r, r)$ for $1 \leq r \leq W$, $1 \leq r \leq k$ and all nodes N of the 1-bit trie O . You should use Eqs. 12.5 through 12.8. Your function should compute $S(N, r)$, which is the s value that minimizes the right side of Eq. 12.5, as well. The complexity of your function should be $O(nW^2k)$, where n is the number of nodes in the router table. Show that this is the case.
- (b) Write a C++ function that determines the strides of all nodes in the best k -VST for O . Your function should use the S values computed in part (a). The complexity of your function should be $O(nW)$. Show that this is the case.

12.6 REFERENCES AND SELECTED READINGS

Digital search trees were first proposed by E. Coffman and J. Eve in *CACM*, 13, 1970, pp. 427-432. The structure Patricia was developed by D. Morrison. Digital search trees, tries, and Patricia are analyzed in the book *The Art of Computer Programming: Sorting and Searching*, Second Edition, by D. Knuth, Addison-Wesley, Reading, MA, 1998.

You can learn more about the genome project and genomic applications of pattern matching from the following Web sites: <http://www.nhgri.nih.gov/HGP/> (NIH's Web site for the human genome project); http://www.ornl.gov/TechResources/Human_Genome/home.html (Department of Energy's Web site for the human genomics project); and <http://merlin.mtbc.bcm.tmc.edu:8001/bcd/Curric/welcome.html>; (Biocomputing Hypertext Coursebook).

Linear time algorithms to search for a single pattern in a given string can be found in most algorithm's texts. See, for example, the texts: *Computer Algorithms*, by E. Horowitz, S. Sahni, and S. Rajasekaran, Computer Science Press, New York, 1998 and *Introduction to Algorithms*, by T. Cormen, C. Leiserson, and R. Rivest, McGraw-Hill Book Company, New York, 1992.

For more on suffix tree construction, see the papers: "A space economical suffix tree construction algorithm," by E. McCreight, *Journal of the ACM*, 23, 2, 1976, 262-272; "Fast string searching with suffix trees," by M. Nelson, *Dr. Dobbs's Journal*, August 1996. and "Suffix trees and suffix arrays," by S. Aluru, in *Handbook of data structures and applications*, D. Mehta and S. Sahni, editors, Chapman & Hall/CRC, 2005.

You can download C++ code to construct a suffix tree from <http://www.ddj.com/http/1996/1996.08/suffix.zip>.

The use of fixed- and variable-stride tries for IP router tables was first proposed in the paper "Faster IP lookups using controlled prefix expansion," by V.

Srinivasan and G. Varghese, *ACM Transactions on Computer Systems*, Feb., 1999. Our dynamic programming formulations for fixed- and variable-stride tries are from "Efficient construction of multi-bit tries for IP lookup," by S. Sahni and K. Kim, *IEEE/ACM Transactions on Networking*, 2003. For more on data structures for IP router tables and packet classification, see "IP router tables," by S. Sahni, K. Kim and H. Lu and "Multi-dimensional packet classification," by P. Gupta, in *Handbook of data structures and applications*, D. Mehta and S. Sahni, editors, Chapman & Hall/CRC, 2003.

